Behavioural Subtyping for a Type-Theoretic Model of Objects

Erik Poll

E.Poll@ukc.ac.uk

Computing Laboratory, University of Kent, Canterbury, UK

Abstract

We present a refinement of the existential object model of Pierce and Turner [PT94]. In addition to signatures (or interfaces) as the types of objects, we also provide *classes* as the types of objects. These class types not only specify an interface, but also a particular implementation.

We show that class types can be interpreted in the standard PER model. Our main result is that the standard interpretation of subtyping in PER models – i.e. subtypes are subpers – is then *behavioural subtyping* in the sense of Leavens [Lea90].

1 Introduction

Like most descriptions of objects in typed lambda calculus or typed object calculus, the existential object model of Pierce and Turner [PT94] provides *signatures* or *interfaces* as the types of objects, and provides the usual syntactic notion of subtyping on these types. We consider a more refined notion of *class type* as the type of an object. A class type is a subtype of an interface type, specified by a particular implementation and initial state.

The existential object encoding can be carried out in $F^{\omega}_{\leq}$, but the class types we want to consider are not encodable in $F^{\omega}_{\leq}$. Therefore we will define a simple object-oriented functional language λ^{OO}, that is essentially just some syntactic sugar for the existential object encoding, but extended with a notion of class type.

The standard model of $F^{\omega}_{\leq}$ is a PER model, which interprets types as partial equivalence relations (pers), and interprets subtyping as subset inclusion between pers [BL90]. We show that class types can be interpreted in the PER model. For these interpretations of class types the subset relation turns out to be equivalent to the notion of *behavioural subtyping* as defined by Leavens [Lea90].

It is important to note that class types can be behavioural subtypes even though they give completely unrelated implementations. Although class types correspond to particular implementations, behavioural subtyping between class types only concerns the observable behaviour of these implementations.

The interpretation of class types can also be viewed from the categorical perspective used in [Rei95] and [HP95]: class types are interpreted as sub-coalgebras of final coalgebras in the PER model.

We briefly review the different notions of (sub)typing for objects, and fix the terminology used for them in this paper.

Interface Types vs Class Types

There are at least two kinds of types that can be used for objects, which we will call *interface types* and *class types.* An *interface type* just specifies an interface (or signature), i.e. it lists the methods with their input and output types. A *class type* not only specifies the signatures of the methods, but also an implementation of the methods and an initial state. An important consequence of this is that the objects of a class type can be guaranteed to have some behaviour in common. The objects of an interface type on the other hand cannot be guaranteed to have any behaviour in common.

Most object-oriented languages provide classes as types. Most type-theoretical work on OO on the other hand concerns interface types (e.g. [Car88], many of the papers in [GM94], [Bru94], [FM94], [AC96]). A notable exception is [FM97], which gives an extensive comparison of interface types and class types.

Signature Subtyping vs Behavioural Subtyping

There are at least two notions of subtyping for object types, which we will call *signature subtyping* and *behavioural subtyping.*

Signature subtyping is a purely syntactic notion, defined by the usual contra/covariant rules (e.g. [Car88]). It concerns just the interfaces of objects: a signature subtype provides (at least) all the methods of the supertype with compatible signatures. Signature subtyping prevents type errors ("message not understood") from occurring at run-time: if A is a signature subtype of B then substituting A's for B's will not cause any type errors.

Behavioural subtyping is a more semantic notion, and concerns more than just signatures. It also tries to capture the intuition that objects in the subtype "behave like" objects in the supertype. It can informally be defined as follows: A is a behavioural subtype of B iff using objects of

To appear in the proceedings of the Fifth International Workshop on the Foundations of Object-Oriented Languages (FOOL5), January 1998, San Diego, USA.

type A in place of objects of type B does not cause any unexpected behaviour. Or, A is a behavioural subtype of B iff any property that holds for all objects of type B also holds for all objects of type A. Another formulation is known as Liskov's substitution principle [Lis88]: "A is a behavioural subtype of B iff for every object a of type A there is an object b of type B such that for all programs p that use b, the behaviour of p is unchanged when b is replaced with a".

Behavioural subtyping is clearly a useful property for reasoning about programs. Behavioural subtyping is in general not decidable, unlike signature behavioural subtyping, which can be statically enforced by a typechecker.

There are different ways to give formal definitions of behavioural subtyping.

One way to define behavioural subtyping is to require there exists a simulation relation between states of objects in the subtype and states of objects in the supertype such that for every object in the behavioural subtype there is an object in the supertype with a related state. This characterisation is the one we will use. It was introduce by Leavens in [Lea90], and is also used in [LW95] [Mau95].

Another – more common – way to define behavioural subtyping is in terms of pre- and post-conditions of methods, and require that methods in a behavioural subtype have weaker pre-conditions and stronger post-conditions than the corresponding methods in the supertypes, so that behavioural subtypes correspond to stronger specifications. This is supported to a certain extent in the programming language Eiffel [Mey88], and is widely used in the literature, e.g. [Ame89] [Lea90] [LW95] [LW94] [PH97] [AL97]. This second approach can combined with the first in order to cope with pre- and postconditions that refer to the abstract values of objects. For example, suppose objects of type $List_A$ are specified in terms of the sequences in $A*$ they represent, and objects of type Set_A in terms of the sets in $\mathcal{P}(A)$ they represent. Then a (simulation) relation $R \subseteq Set_A \times \mathcal{P}(A)$ could be used to relate these specifications. This approach is used in [Lea90]. It is also used in [Ame89][LW94], where simulation relations are restricted to functions.

The rest of this paper is organised as follows. Section 2 gives some examples to illustrate the notions mentioned above. Section 3 then gives a formal definition of λ^{OO}. Section 4 discusses the PER model for $F_{\leq}$ of [PAC94], which provides a PER model for λ^{OO} without class types. This PER model is then used in Section 5 as the basis of a PER model for λ^{OO} including the class types, and we show that subtyping between class types in this model is behavioural subtyping. We conclude in Section 6.

2 Informal Introduction to λ^{OO}

We give some simple examples to introduce λ^{OO} and the existential object model, and to explain the notions of interface types, class types, signature subtyping, and behavioural subtyping in more detail.

2.1 Objects and Interface Types

The interface type

$$Counter \quad = \quad \mathbf{Sig}\,(X)\{getcount : Nat, count : X\}$$

is the type of objects with methods *getcount* and *count*, where *getcount* returns a natural number and *count* a new object of the same type. We write $o \leftarrow l$ for the invocation of method l of object o. So, if $o : Counter$ then $o \leftarrow getcount : Nat$ and $o \leftarrow count : Counter$.

An object of type $Counter$ – a counter – can be constructed from a state s of some type Rep and a method table $m : Rep \rightarrow CounterI(Rep)$ that gives the implementation of the methods, where

$$CounterI(X) \quad = \quad \{getcount : Nat, count : X\}.$$

This object is written as **object** $\langle s, m \rangle$.

For example, we could use the type $\{x : Nat\}$ – the type of records with an x-field of type Nat – to represent the state of counter, and use the following function as method table

$$\begin{array}{lll} m & = & \lambda s : \{x : Nat\}.\, \{get = s.x, count = \{x = s.x + 1\}\} \\ & : & \{x : Nat\} \rightarrow CounterI(\{x : Nat\}) \end{array}$$

E.g. **object** $\langle \{x = 5\}, m \rangle$ is the counter-object with $\{x = 5\}$ as state and m as method table.

A simple operational semantics for method invocation can be given as follows

$$\begin{array}{rcl} (\mathbf{object}\ \langle s, m \rangle) \leftarrow getcount & = & (m\ s).getcount \\ (\mathbf{object}\ \langle s, m \rangle) \leftarrow count & = & \mathbf{object}\ \langle (m\ s).count, m \rangle \end{array}$$

For example, if $o \equiv \mathbf{object}\langle \{x = 5\}, m \rangle$, then $o \leftarrow getcount = 5$ and $o \leftarrow count = \mathbf{object}\ \langle \{x = 6\}, m \rangle$. Note that the methods *count* and *getcount* are treated differently, because *count* returns an object of the same class and *getcount* just returns a natural number. We will call a method such as *count* a *mutator* and a method such as *getcount* an *observer*. In general, to invoke a method l we apply the method table to the state and select l component, and, if the method is mutator, we wrap up the new state with the old method table to produce an object.

Everything described so far is just syntactic sugar for the existential object encoding of [PT94]. We have written $\mathbf{Sig}\ CounterI$ for

$$\exists Rep.Rep \times (Rep \rightarrow CounterI(Rep))$$

and **object** $\langle s, m \rangle$ for

$$\mathbf{pack}\ \langle Rep, (s, m) \rangle\ \mathbf{to}\ \exists Rep.Rep \times (Rep \rightarrow CounterI(Rep))$$

where Rep is the type of s.

Existential types model abstract types [MP88]. The existential type above hides the type Rep that used to represent the state of an object, so that the state s of an object **object** $\langle s, m \rangle$ can only be observed indirectly by invoking the methods *getcount* and *count*.

In certain models – including the PER model we will use – the existential type above is interpreted as a final coalgebra [PA93][Has94]. Then equality of objects is *bisimulation*: **object** $\langle s_1, m_1 \rangle$ and **object** $\langle s_2, m_2 \rangle$ of type $Counter$ are equal if their states are related by some simulation relation $\sim$, i.e. if $s_1 \sim s_2$ for some relation $\sim$ such that

$$x_1 \sim x_2 \Rightarrow \left\{ \begin{array}{c} (m_1\, x_1).getcount =_{Nat} (m_2\, x_2).getcount \\ (m_2\, x_1).count \sim (m_2\, x_2).count \end{array} \right.$$

2.2 Subtyping on Interface Types

An example of a signature subtype of *Counter* is

$$RCounter \quad = \quad \mathbf{Sig}\, RCounterI,$$

where

$$RCounterI(X) \quad = \quad \{getcount : Nat, count : X, reset : X\}.$$

RCounter is type of counters that in addition to methods *count* and *getcount* also have a method *count*.

The PER model does indeed interpret *RCounter* and *Counter* as subsets: $[\![RCounter]\!] \subseteq [\![Counter]\!]$. We can think of the partial equivalence relations $[\![RCounter]\!]$ and $[\![Counter]\!]$ as the notion of equality for counters and resetable counters, respectively. Then $[\![RCounter]\!] \subseteq [\![Counter]\!]$ means that $[\![Counter]\!]$ is a coarser notion of equality than $[\![RCounter]\!]$. This can be understood as follows. Suppose we have two objects that can not be distinguished by invoking just the methods *count* and *getcount*, but that can be distinguished by invoking the methods *count*, *getcount* and *reset*. These objects would then be identified by $[\![Counter]\!]$, but would not be identified by $[\![RCounter]\!]$.

The subtyping between *RCounter* and *Counter* is a degenerated instance of behavioural subtyping. For interface types such as *RCounter* and *Counter* the notions of signature subtyping and behavioural subtyping are identical, because these types do not make any behavioural guarantees.

2.3 Class Types

The class types we consider not only specify an interface, but also fix the type used for the representation, the implementation of the methods, and an initial state. An example of a class type is

$$CounterClass \quad = \quad (\mathbf{Class}\, CounterI\, \mathbf{with}\, init, m)$$

where

$$\begin{array}{rcl} init & = & \{x = 1\} \\ m & = & \lambda s{:}\{x : Nat\}.\, \{get = s.x, count = \{x = s.x + 1\}\} \end{array}$$

Intuitively, *CounterClass* is the following inductively defined set:

- $\mathbf{object}\ \langle init, m\rangle : CounterClass$,
- if $c : CounterClass$ then $c{\leftarrow}count : CounterClass$.

i.e. *CounterClass* can be understood as the smallest subset of *Counter* that contains $\mathbf{object}\ \langle init, m\rangle$ and is closed under method invocation.

CounterClass is like a class definition in a standard OO language, in that it introduces a type together with an implementation and initialisation for the objects of that type. In a programming language we might write something like "*new.CounterClass*" for $\mathbf{object}\ \langle init, m\rangle$.

Because the objects in *CounterClass* all have m as their method table and all have a hidden state of type $\{x : Nat\}$ that is "reachable" from the initial state s by method invocations, they have some behaviour in common. For example,

$$o{\leftarrow}getcount \geq 1$$

and

$$(o{\leftarrow}count){\leftarrow}getcount = o{\leftarrow}getcount + 1$$

for all $o : CounterClass$.

Interpreting class types in the PER model is not a problem, and we get the expected relation between the pers interpreting *CounterClass* and *Counter*: $[\![CounterClass]\!] \subseteq [\![Counter]\!]$. The pers $[\![CounterClass]\!]$ and $[\![Counter]\!]$ provide the same notion of equality, i.e. all $[\![CounterClass]\!]$ equivalence classes are also $[\![Counter]\!]$equivalence classes, but there are fewer $[\![CounterClass]\!]$ equivalence classes than there are $[\![Counter]\!]$equivalence classes.

2.4 Subtyping on Class Types

Assume there is a type *natlist* of lists of natural numbers, with the usual constructors *nil* and *cons*. The class type below uses a state of type $\{y : natlist\}$ to represent the state of a resetable counter:

$$RCounterClass = (\mathbf{Class}\, RCounterI\, \mathbf{with}\, init_R, m_R)$$

where

$$\begin{array}{rcl} init_R & = & \{y = cons\ 0\ (cons\ 0\ nil)\} \\ m_R & = & \lambda s{:}\{y : natlist\}. \\ & & \quad \{getcount = length(s.y) \\ & & \quad ,count = \{y = cons\ 0\ s.y\} \\ & & \quad ,reset = \{y = cons\ 2\ (cons\ 2\ nil)\}\} \end{array}$$

RCounterClass can be understood as the smallest subset of *RCounter* that contains $\mathbf{object}\ \langle init_R, m_R\rangle$ and is closed under method invocation.

RCounterClass can be viewed as a behavioural subtype of *CounterClass*, because, despite their different implementations, the objects in *RCounterClass* behave just like objects in *CounterClass*. By invoking only the methods *count* and *getcount*, it is not possible to distinguish the objects in *RCounterClass* from those in *CounterClass*, since for every object $o' : RCounterClass$ there is a $o : CounterClass$ that is indistinguishable from o'. More formally, we can say that *RCounterClass* is a behavioural subtype of *CounterClass* because there exists a relation $\sim \subseteq \{x : Nat\} \times \{y : natlist\}$ such that

$$s \sim s' \Rightarrow \left\{ \begin{array}{c} (m\ s).getcount =_{Nat} (m_R\ s').getcount \\ (m\ s).count \sim (m_R\ s').count \end{array} \right. \qquad \text{(i)}$$

and

$$\begin{array}{l} \forall o'{:}RCounterClass. \\ \ \exists o{:}CounterClass. \\ \ \ \text{the states of } o \text{ and } o' \text{ are related by } \sim. \end{array} \qquad \text{(ii)}$$

This relation $\sim$ is of course

$$s \sim s' \iff s.x = length(s'.y)\ .$$

Together conditions (i) and (ii) guarantee that for every $o' : RCounterClass$ there is a $o' : CounterClass$ such that o' is indistinguishable from o, if we are only allowed to invoke their *getcount* and *count* methods.

Intuitively, the conditions above guarantee that objects in *RCounterClass* have the properties that all objects in *CounterClass* have, and maybe more. For instance, note that $o{\leftarrow}getcount \geq 2$ for all $o : RCounterClass$.

Examples of classes that are not behavioural subtypes of *CounterClass* are

$$\begin{array}{rcl} CounterClass1 & = & (\mathbf{Class}\, CounterI\, \mathbf{with}\, init, m') \\ RCounterClass1 & = & (\mathbf{Class}\, CounterI\, \mathbf{with}\, init_R, m'_R) \end{array}$$

with

$$
\begin{array}{rcl}
m' & = & \lambda s{:}\{x : Nat\}.\, \{get = s.x, count = \{x = s.x + 2\}\} \\
m'_R & = & \lambda s{:}\{y : natlist\}. \\
 & & \quad \{getcount = length(s.y) \\
 & & \quad , count = \{y = cons\ 0\ s.y\} \\
 & & \quad , reset = \{y = nil\}\}
\end{array}
$$

These classes are not behavioural subtypes of $CounterClass$, for slightly different reasons. For $CounterClass1$ we cannot find a suitable simulation relation. For $RCounterClass1$ there is an obvious simulation relation, namely the same one we used for $RCounterClass$. However, there is a object in $RCounterClass1$ for which there is no related object in $CounterClass$: resetting a counter in $RCounterClass1$ produces an object $o \equiv \mathbf{object}\ \langle \{y = nil\}, m\rangle$ for which $o{\leftarrow}getcount = 0$, and this object cannot be related to any object in $CounterClass$, since all counters in $CounterClass$ have a $getcount$ greater then 0.

For class types, the interpretation of subtyping in the PER model – subtypes are subpers – turns out to be equivalent to the informal notion of behavioural subtyping introduced here, i.e. given by properties such as (i) and (ii). So in the PER model

$$
\begin{array}{rcl}
[\![RCounterClass]\!] & \subseteq & [\![Counterclass]\!] \\
[\![CounterClass1]\!] & \not\subseteq & [\![Counterclass]\!] \\
[\![RCounterClass1]\!] & \not\subseteq & [\![Counterclass]\!]
\end{array}
$$

The fact that the PER model already provides bisimulation as the notion of equality plays a vital role here.

3 Definition of λ^{OO}

The raw syntax of the *terms* a, *types* A and *signatures* I of λ^{OO} is given by the following grammar

$$
\begin{array}{rcl}
a & ::= & x \mid \lambda x{:}A.\, a \mid aa \mid \{l_1 = a, \ldots, l_n = a\} \mid a.l \mid \\
 & & a{\leftarrow}l(a) \mid \mathbf{object}_I\ \langle A, a, a\rangle \\
A & ::= & X \mid A{\rightarrow}A' \mid \{l_1{:}A, \ldots, l_n{:}A\} \mid \\
 & & \mathbf{Sig}\ I \mid \mathbf{Class}\ I\ \mathbf{with}\ A, a, a \\
I & ::= & (X)\{l_1{:}A{\rightarrow}A, \ldots, l_n{:}A{\rightarrow}A\}
\end{array}
$$

Here x ranges over *term variables*, X ranges over *type variables*, and l over a countable set $\mathcal{L}$ of *labels*. The type variable X is bound in $(X)A$, and if $I \equiv (X)A$ we write $I(B)$ for $[B/X]A$, where $[B/X]A$ denotes the capture-free substitution of B for X in A. Expressions equal up to the names of bound variables and permutation of fields are identified as usual, and we assume that the same label never occurs twice in a record (type) or interface.

For simplicity, we divide $\mathcal{L}$ into a set $\mathcal{L}_{obs}$ of labels that can be used as names for *observer methods* and a set $\mathcal{L}_{mut}$ of labels that can be used as names for *mutator methods*, and we insist that

$$
\begin{array}{rcl}
l_i \in \mathcal{L}_{obs} & \Rightarrow & X \notin \mathsf{FV}(A_i{\rightarrow}B_i) \\
l_i \in \mathcal{L}_{mut} & \Rightarrow & X \notin \mathsf{FV}(A_i) \wedge X \equiv B_i
\end{array}
$$

for any signature $(X)\{l_1{:}A_1{\rightarrow}B_1, , \ldots, l_n{:}A_n{\rightarrow}B_n\}$.

The *contexts* of λ^{OO} are given by

$$
\Gamma \quad ::= \quad \epsilon \mid \Gamma, x : A
$$

with no variable occurring twice.

The subtyping and typing rules of λ^{OO} are given in tables 1 and 3. The rules for well-formedness of contexts and types are given in Table 2. These are needed because there are types with terms as subexpressions, namely the class types. Only the well-formedness rule for class types is given, for the other are trivial. (E.g., $A{\rightarrow}B$ is well-formed in Γ if A and B are well-formed in Γ, etc.)

The *reduction relation* $\rhd$ on terms is given by the rules

$$
\begin{array}{rcl}
(\lambda x{:}A.\, b)a & \rhd & [a/x]b \\
\{l_1 = a_1, \ldots, l_n = a_n\}.l_i & \rhd & a_i \\
(\mathbf{object}_I\ \langle s, m\rangle){\leftarrow}l(a) & \rhd & (m\ s).l\ a \\
 & & \text{if } l \in \mathcal{L}_{obs} \\
(\mathbf{object}_I\ \langle s, m\rangle){\leftarrow}l(a) & \rhd & \mathbf{object}_I\ \langle (m\ s).l\ a, m\rangle \\
 & & \text{if } l \in \mathcal{L}_{mut}
\end{array}
$$

where $[a/x]b$ denotes the capture-free substitution of a for x in b.

We write $\simeq$ for the reflexive, transitive, and symmetric closure of $\rhd$, and $\Gamma \vdash a \simeq a' : A$ for $\Gamma \vdash a : A \wedge \Gamma \vdash a' : A \wedge a \simeq a'$.

4 PER Model for λ^{OO} without Class Types

As we mentioned earlier, with the exception of the class types, λ^{OO} just introduces some syntactical sugar for the existential object encoding of objects in $F^\omega_\leq$ given in [PT94]. In fact, we only need the subsystem $F_\leq$ of $F^\omega_\leq$ as target system.[1] So any model of $F_\leq$ can be used as model for λ^{OO} without classes.

The model we use is the PER model of [PAC94]. This model combines the PER models of [BL90] and [BFAS90]: it is essentially just the model of [BL90] – and interprets subtypes as subpers –, but it uses the interpretation of polymorphic types given in [BFAS90] – rather than the more standard one used in [BL90] – to ensure parametricity. The vital property of this model that we are interested in is that the existential types $\exists X.\, X \times (X{\rightarrow}I(X))$ are interpreted as *final coalgebras*, as is proved in [Has94] and [PA93].

First we make precise how λ^{OO} without class types can be regarded as syntactic sugar for $F_\leq$.

As far as types are concerned the types, interface types abbreviate existential types:

$$
\mathbf{Sig}\ I \quad = \quad \exists X.\, X \times (X{\rightarrow}I(X)).
$$

Remark 4.1 Here $\times$ and $\exists$ stand for the usual $F_\leq$ encodings. The record types we have in λ^{OO} are not part of $F_\leq$, but these can be encoded in $F_\leq$ using the trick of [Car92]: We assume there is some enumeration of the labels and we represent the record type $\{l_1 : A_1, \ldots, l_n : A_n\}$ by the product $B_1 \times \ldots B_m$ where m is the greatest index of the l_i and $B_j = A_i$ if j is the index of l_i and $B = Top$ if j is not the index of any of the l_i. Product types can of course be encoded in $F_\leq$ in the usual way.

The encodings of existential types, product types and record types will be left implicit, so we use the normal notation for existentials, products, and records as abbreviation for their $F_\leq$-encodings.

[1] In [PT94] $F^\omega_\leq$ rather than $F_\leq$ is needed to type generic method invocations, i.e. method invocations not yet applied to a particular object, which in our syntax would be written as "$\leftarrow l$". We don't have these in λ^{OO}.

$$\frac{}{A \leq A}\ \leq\text{-refl} \qquad \frac{A \leq A' \quad A' \leq A''}{A \leq A''}\ \leq\text{-trans} \qquad \frac{A_1 \leq B_1 \dots A_n \leq B_n \quad m \geq n}{\{l_1{:}A_1, \dots, l_n{:}A_m\} \leq \{l_1{:}B_1, \dots, l_n{:}B_n\}}\ \leq\text{-record}$$

$$\frac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'}\ \leq\text{-}\to \qquad \frac{A \leq B}{\mathbf{Sig}\ (X)A \leq \mathbf{Sig}\ (X)B}\ \leq\text{-sig} \qquad \frac{}{(\mathbf{Class}\ I\ \mathbf{with}\ meth, init) \leq \mathbf{Sig}\ I}\ \leq\text{-class}$$

Table 1. Subtyping

$$\frac{}{\epsilon \vdash ok}\ \text{empty-ok} \qquad \frac{\Gamma \vdash ok \quad \Gamma \vdash A\ ok}{\Gamma, x : A \vdash ok}\ \text{weaken-ok} \qquad \frac{\Gamma ok \quad \Gamma \vdash init : Rep \quad \Gamma \vdash m : Rep \to I(Rep)}{\Gamma \vdash (\mathbf{Class}\ I\ \mathbf{with}\ s, m)\ ok}\ \text{class-ok}$$

Table 2. Well-formedness

$$\frac{\Gamma, x : A, \Gamma' \vdash ok}{\Gamma, x : A, \Gamma' \vdash x : A}\ \text{var} \qquad \frac{\Gamma \vdash a : A \quad A \leq B}{\Gamma \vdash a : B}\ \text{sub} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.\, b : A \to B}\ \to\text{-I} \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B}\ \to\text{-E}$$

$$\frac{\Gamma \vdash a_1 : A_1 \quad \dots \quad \Gamma \vdash a_n : A_n}{\Gamma \vdash \{l_1 = a_1, \dots, l_n = a_n\} : \{l_1{:}A_1, \dots, l_n{:}A_n\}}\ \text{record-I} \qquad \frac{\Gamma \vdash a : \{l_1{:}A_1, \dots, l_n{:}A_n\} \quad l_i \in \{l_1, \dots, l_n\}}{\Gamma \vdash a.l_i : A_i}\ \text{record-E}$$

$$\frac{\Gamma \vdash s : Rep \quad \Gamma \vdash m : Rep \to I(Rep)}{\Gamma \vdash \mathbf{object}_I\ \langle s, m\rangle : \mathbf{Sig}\ I}\ \text{object-I1} \qquad \frac{\Gamma \vdash s : Rep \quad \Gamma \vdash m : Rep \to I(Rep)}{\Gamma \vdash \mathbf{object}_I\ \langle s, m\rangle : (\mathbf{Class}\ I\ \mathbf{with}\ s, m)}\ \text{object-I2}$$

For $SC \equiv \mathbf{Sig}\ I$ or $SC \equiv (\mathbf{Class}\ I\ \mathbf{with}\ meth, init)$ with $I(X) = \{\dots, l : A \to B, \dots\}$:

$$\frac{\Gamma \vdash o : SC \quad \Gamma \vdash a : A \quad l \in \mathcal{L}_{obs}}{\Gamma \vdash o \leftarrow l(a) : B}\ \text{object-E-obs} \qquad \frac{\Gamma \vdash o : SC \quad \Gamma \vdash a : A \quad l \in \mathcal{L}_{mut}}{\Gamma \vdash o \leftarrow l(a) : SC}\ \text{object-E-mut}$$

Table 3. Typing

The λ^{OO} syntax for object formation and method invocation can be interpreted by the following $F_{\leq}$-terms:

$$\begin{array}{rcl}
\mathbf{object}_I\ \langle s, m\rangle & = & \mathbf{pack}\ \langle Rep, (s, m)\rangle\ \mathbf{to}\ \mathbf{Sig}\ I \\
o \leftarrow l(a) & = & \mathbf{open}\ o\ \mathbf{as}\ \langle X, (s, m)\rangle\ \mathbf{in}\ (m\ s).l\ a \\
& & \quad \text{if } l \in \mathcal{L}_{obs} \\
o \leftarrow l(a) & = & \mathbf{open}\ o\ \mathbf{as}\ \langle X, (s, m)\rangle\ \mathbf{in} \\
& & \quad (\mathbf{pack}\ \langle X, (m\ s).l\ a, m)\rangle\ \mathbf{to}\ \mathbf{Sig}\ I) \\
& & \quad \text{if } l \in \mathcal{L}_{mut}
\end{array}$$

Note that some type information is missing in the λ^{OO}-terms, namely the type Rep of s in the first clause and $\mathbf{Sig}\, I$ in the last clause. However, type information in terms does not play a role in the interpretation of terms in the PER model that we are interested in, so we can safely ignore this.

We now consider the PER model of [PAC94]. Here types are interpreted as partial equivalence relations (pers) on $\mathbb{N}$, and terms are interpreted as natural numbers, using some enumeration of the partial recursive functions.

The per interpreting a type A is written as $[\![A]\!]_\xi$, where ξ is a type environment that maps from type variables to pers. The natural number interpreting a term a is written as $[\![a]\!]_\eta$, where η is a term environment that maps terms to natural numbers. Another way of looking at the PER model is to interpret a type A as the set of $[\![A]\!]_\xi$-equivalence classes, and to interpret a term $a : A$ as the $[\![A]\!]_\xi$-equivalence class that contains $[\![a]\!]_\eta$.

λ^{OO} without class types can be interpreted in this PER model for $F_{\leq}$ by interpreting interface types, objects, and method invocations by their $F_{\leq}$-counterparts:

$$\begin{array}{rcl}
[\![\mathbf{Sig}\ I]\!]_\xi & = & [\![\exists X.\, X \times (X \to I(X))]\!]_\xi \\
[\![\mathbf{object}_I\ \langle s, m\rangle]\!]_\eta & = & [\![\mathbf{pack}\ \langle Rep, (s, m)\rangle\ \mathbf{to}\ \mathbf{Sig}\ I]\!]_\eta \\
[\![o \leftarrow l(a)]\!]_\eta & = & [\![\mathbf{open}\ o\ \mathbf{as}\ \langle X, (s, m)\rangle\ \mathbf{in}\ (m\ s).l\ a]\!]_\eta \\
& & \quad \text{if } l \in \mathcal{L}_{obs} \\
[\![o \leftarrow l(a)]\!]_\eta & = & [\![\mathbf{open}\ o\ \mathbf{as}\ \langle X, (s, m)\rangle\ \mathbf{in} \\
& & \quad \mathbf{pack}\ \langle X, (m\ s).l\ a, m)\rangle\ \mathbf{to}\ \mathbf{Sig}\ I]\!]_\eta \\
& & \quad \text{if } l \in \mathcal{L}_{mut}
\end{array}$$

A pair (η, ξ) satisfies a context Γ – written $(\eta, \xi) \models \Gamma$ – iff $(\eta(x), \eta(x)) \in [\![A]\!]_\xi$ for all declarations $x : A$ in Γ. Given that the PER model is a sound model for $F_{\leq}$, it is also a sound model for λ^{OO}:

Theorem 4.2
If $\Gamma \vdash a \simeq a' : A$ in λ^{OO} without class types, then $([\![a]\!]_\eta, [\![a']\!]_\eta) \in [\![A]\!]_{\eta,\xi}$ for all $(\eta, \xi) \models \Gamma$.

We will now look at the interpretations of objects and interface types in more detail. Some properties of these interpretations will be used to interpret class types in the next section.

First a few definitions. We write $n \cdot m$ for the n^{th} partial recursive function applied to m, and $\lambda\!\!\lambda x.\, E(x)$ for the index of a partial recursive function for which $\lambda\!\!\lambda x.\, E(x) \cdot n = E(n)$, where $E(x)$ is a partial recursive description of a natural number depending on x.

We write $[n]_R$ for the R-equivalence containing n, and $\mathbb{N}/R$ for the set of R-equivalence classes. For pers R and S, the per $R \twoheadrightarrow S$ is defined as

$$R \twoheadrightarrow S = \{(f, f') \in \mathbb{N} \times \mathbb{N} \mid \forall (a, a') \in A.\, (f \cdot a, f' \cdot a') \in B\}.$$

We write $n \sqsubseteq R$ as abbreviation for $(n,n) \in R$.

The category PER is defined as in [BFAS90] and [Has94] as the category with as objects pers on $\mathbb{N}$ and as the arrows from R to S total functions from $\mathbb{N}/R$ to $\mathbb{N}/S$ named by a partial recursive functions in $R \twoheadrightarrow S$.

4.1 Interpretation of objects

We define $\mathsf{obj}\langle\_,\_\rangle$ and $\_ \Leftarrow_l \_$ as the interpretations of **object** $\langle\_,\_\rangle$ and $\_{\leftarrow}l(\_)$ in the PER model. So

$$[\![\mathbf{object}_I\ \langle s,m\rangle]\!]_\eta \quad = \quad \mathsf{obj}\langle [\![s]\!]_\eta, [\![m]\!]_\eta\rangle$$

and

$$[\![o{\leftarrow}l(a)]\!]_\eta \quad = \quad [\![o]\!]_\eta \Leftarrow_l [\![a]\!]_\eta \ .$$

Looking at the interpretations of the $F_{\leq}$-terms that **object** $\langle\rangle$ and $\leftarrow l$ denote, we find that

$$(\mathsf{obj}\langle s,m\rangle) \Leftarrow_l a \quad = \quad \begin{cases} (m\cdot s).l\cdot a & \text{if } l\in\mathcal{L}_{obs} \\ \mathsf{obj}\langle (m\cdot s).l\cdot a, m\rangle & \text{if } l \in \mathcal{L}_{mut}\end{cases}$$

Note that we abuse our notation for field selection here and write ".l" for the selection of the l-field, when we should really use the interpretation of the l-projection under the encoding of records as products discussed in Remark 4.1.

4.2 Interpretation of interface types

The interesting property of the PER model is that the existential types of the form $\exists X.\, X\times(X{\rightarrow}I(X))$ – i.e. interface types – are interpreted as final coalgebras, as is proved in [Has94] and [PA93].

We define $\mathsf{SIG}(\_)$ as the interpretation of **Sig** $(\_)$ in the PER model. So

$$\begin{aligned}[\![\mathbf{Sig}\ I]\!]_\xi \quad &= \quad [\![\exists X.\, X\times(X{\rightarrow}I(X))]\!]_\xi \\ &= \quad \mathsf{SIG}([\![I]\!]_\xi)\end{aligned}$$

Let $\mathcal{I} : \mathsf{PER}{\rightarrow}\mathsf{PER}$ be the functor interpreting some interface I in some environment ξ, i.e. $\mathcal{I} = [\![I]\!]_\xi$. We define $\mathsf{out}_\mathcal{I}$ as

$$\mathsf{out}_\mathcal{I} \quad = \quad \lambda\!\!\lambda o.\, \mathcal{I}(\lambda\!\!\lambda s.\, \mathsf{obj}\langle s, \mathsf{snd}(o)\rangle)\cdot(\mathsf{snd}(o)\cdot\mathsf{fst}(o)) \ .$$

Another way of defining $\mathsf{out}_\mathcal{I}$ is as the interpretation of out_I : **Sig** $I \rightarrow I($**Sig** $I)$, defined as follows

$$\begin{aligned} out_I \quad &= \quad \lambda o{:}\mathbf{Sig}\ I.\ \mathbf{open}\ o\ \mathbf{as}\ \langle X,(s,m)\rangle\ \mathbf{in} \\ &\qquad\quad I_m(\lambda x{:}X.\ \mathbf{object}_I\ \langle x,m\rangle)\ (m\ s) \\ &: \quad \mathbf{Sig}\ I \rightarrow I(\mathbf{Sig}\ I)\end{aligned}$$

Here I_m is the action of I on functions, with $I_m(f) : I(A){\rightarrow}I(B)$ for any $f : A{\rightarrow}B$, defined in the usual way by induction on I.

The pair $(\mathsf{SIG}(\mathcal{I}), \mathsf{out}_\mathcal{I})$ is a final coalgebra [Has94][PA93]. So

$$\mathsf{out}_\mathcal{I} \sqsubseteq \mathsf{SIG}(\mathcal{I}) \twoheadrightarrow \mathcal{I}(\mathsf{SIG}(\mathcal{I})),$$

and we have the following properties;

Property 4.3 *Let $s_i \sqsubseteq Rep_i$ and $m_i \sqsubseteq Rep_i \twoheadrightarrow \mathcal{I}(Rep_i)$ for certain pers Rep_i, $i = 1,2$.*

Then

$$\begin{aligned} & (\mathsf{obj}\langle s_1,m_1\rangle, \mathsf{obj}\langle s_2,m_2\rangle) \in \mathsf{SIG}(\mathcal{I}) \\ \iff & \\ & \exists \sim\in \mathbb{N}\times\mathbb{N}.\ \ \sim = Rep_1;\sim;Rep_2 \ \wedge \\ & \qquad\qquad\quad s_1 \sim s_2 \ \wedge \\ & \qquad\qquad\quad (m_1,m_2) \in \sim\twoheadrightarrow \mathcal{I}(\sim)\end{aligned}$$

Here ; denotes composition of relations.

Property 4.4 *For any relation $\sim\subseteq \mathbb{N}\times\mathbb{N}$*

$$\mathsf{out}_\mathcal{I} \sqsubseteq \sim\twoheadrightarrow\mathcal{I}(\sim) \ \Rightarrow \ \sim\subseteq \mathsf{SIG}(\mathcal{I}).$$

In other words, $\mathsf{SIG}(\mathcal{I})$ is the maximum bisimulation.

These properties are particular cases of Theorems 7 and 11 in [PA93]. A direct consequence (take $\sim = Rep_i = Rep$) of Property 4.3 is:

Corollary 4.5 *Let $(s_1,s_2) \in Rep$ and $(m_1,m_2) \in Rep \twoheadrightarrow I(Rep)$ for some per Rep.*

Then $(\mathsf{obj}\langle s_1,m_1\rangle, \mathsf{obj}\langle s_2,m_2\rangle) \in \mathsf{SIG}(\mathcal{I})$.

The mapping $\mathsf{out}_\mathcal{I}$ is related to the interpretation of method invocations $\_ \Leftarrow_l \_$ as follows:

Property 4.6 *Let $\mathcal{I} = [\![I]\!]_\xi$ for some ξ, with I an interface that contains a method l. Then*

$$o \Leftarrow_l a \quad = \quad (\mathsf{out}_\mathcal{I}\cdot o).l\cdot a$$

As before, we abuse our notation for field selection here.

5 Model for λ^{OO} with Class Types

We now extend the interpretation of λ^{OO} without class types in the PER model of [PAC94] to an interpretation of the full λ^{OO} including the class types.

Definition 5.1 *The relation $\sqsubseteq$ on pers is defined by*

$$R \sqsubseteq S \quad\iff\quad \mathbb{N}/R \subseteq \mathbb{N}/S.$$

An equivalent definition is

$$R \sqsubseteq S \quad\iff\quad R\subseteq S \wedge R = S;R;S \ .$$

The relation $\sqsubseteq$ is used to define the interpretation of a class types:

Definition 5.2 *For $\mathcal{I} : \mathsf{PER}{\rightarrow}\mathsf{PER}$ and $m,n \in \mathbb{N}$ we define $\mathsf{CLASS}(\mathcal{I},s,m)$ as follows*

$$\begin{aligned} & \mathsf{CLASS}(\mathcal{I},s,m) \\ = & \\ & \bigcap\{X \sqsubseteq \mathsf{SIG}(\mathcal{I}) \mid \mathsf{obj}\langle s,m\rangle \sqsubseteq X \wedge \mathsf{out}_\mathcal{I} \sqsubseteq X \twoheadrightarrow \mathcal{I}(X)\}.\end{aligned}$$

This defines $\mathbb{N}/\mathsf{CLASS}(\mathcal{I},s,m)$ as the smallest subset of $\mathbb{N}/\mathsf{SIG}(\mathcal{I})$ that contains $[\mathsf{obj}\langle s,m\rangle]_{\mathsf{SIG}(\mathcal{I})}$ and is closed under method invocations.

Lemma 5.3 *Let $\mathcal{I} : \mathsf{PER}{\rightarrow}\mathsf{PER}$. Suppose that $\mathcal{I}$ is continuous – i.e. $\mathcal{I}(\bigcap_i X_i) = \bigcap_i \mathcal{I}(X_i)$ – and suppose that $\mathcal{I}(R);\mathcal{I}(S) \subseteq \mathcal{I}(R;S)$ for all pers R and S. Then*

1. $\mathsf{CLASS}(\mathcal{I},s,m) \sqsubseteq \mathsf{SIG}(\mathcal{I})$

2. $\mathsf{obj}\langle s,m\rangle \sqsubseteq \mathsf{CLASS}(\mathcal{I},s,m)$
3. $out_I \sqsubseteq \mathsf{CLASS}(\mathcal{I},s,m) \twoheadrightarrow I(\mathsf{CLASS}(\mathcal{I},s,m))$
4. *Let* $(s_1,s_2) \in Rep$ *and* $(m_1,m_2) \in Rep \twoheadrightarrow \mathcal{I}(Rep)$. *Then* $\mathsf{CLASS}(\mathcal{I},s_1,m_1) = \mathsf{CLASS}(\mathcal{I},s_2,m_2)$.

Proof. These properties easily follow from the definition of CLASS and the assumptions on $\mathcal{I}$. For *4.* use Corollary 4.5 to deduce that

$$(s,m) \sqsubseteq X \Longleftrightarrow (s',m') \sqsubseteq X \qquad \text{for any } X \sqsubseteq \mathsf{SIG}(I)$$

from the assumptions on s_i and m_i. □

It is easy to verify that any $\mathcal{I}$ that is the interpretation of a λ^{OO}-signature will satisfy the conditions of the lemma above.

CLASS will now be used to extend the PER model of λ^{OO} without class types to a model for the full λ^{OO}. The definition of this model is given below.

As far as terms is concerned nothing changes. In λ^{OO} without class types we have the same terms as in λ^{OO} with class type, so the terms can be interpreted as in the PER model discussed in the previous section:

Definition 5.4 *The interpretation* $[\![a]\!]_\eta \in \mathbb{N}$ *of a* λ^{OO}*-term* M *in term environment* η *is defined as*

$$[\![a]\!]_\eta = [\![Erase(a)]\!]_\eta\,,$$

where

$$\begin{array}{rcl}
Erase(\mathbf{object}_I\ \langle s,m\rangle) &=& \mathbf{object}\ \langle Erase(s), Erase(m)\rangle \\
Erase(o \leftarrow l(a)) &=& Erase(o) \leftarrow l(Erase(a)) \\
Erase(\lambda x{:}A.\,b) &=& \lambda x.\,Erase(b) \\
Erase(fa) &=& Erase(f)Erase(a) \\
Erase(a.l) &=& Erase(a).l \\
\multicolumn{3}{l}{Erase(\{l_1 = a_1,\ldots,l_n = a_n\}) =} \\
&& \{l_1 = Erase(a_1),\ldots,l_n = Erase(a_n)\}
\end{array}$$

and he interpretation of erased terms is defined by

$$\begin{array}{rcl}
[\![x]\!]_\eta &=& \eta(x) \\
[\![\lambda x.\,b]\!]_\eta &=& \lambda\!\!\lambda n.\ [\![b]\!]_{[n/x]\eta} \\
[\![fa]\!]_\eta &=& [\![f]\!]_\eta \cdot [\![a]\!]_\eta \\
[\![\mathbf{object}_I\ \langle s,m\rangle]\!]_\eta &=& \mathsf{obj}\langle [\![s]\!]_\eta, [\![m]\!]_\eta\rangle \\
[\![o \leftarrow l(a)]\!]_\eta &=& [\![o]\!]_\eta \Leftarrow_l [\![a]\!]_\eta \\
[\![\{l_1 = a_1,\ldots,l_n = a_n\}]\!]_\eta &=& \{l_1 = [\![a_1]\!]_\eta,\ldots,l_n = [\![a_n]\!]_\eta\} \\
[\![a.l]\!]_\eta &=& [\![a]\!]_\eta .l
\end{array}$$

Here we again abuse our notation for records and field selection as shorthand for their interpretations under the encoding discussed in Remark 4.1.

Because class types contain terms as subexpressions, the interpretation of types now has to be given w.r.t. a term environment η as well as a type environment ξ :

Definition 5.5 *The interpretation* $[\![A]\!]_{\eta,\xi}$ *of a type* A *in environment* (η,ξ) *is given by*

$$\begin{array}{rcl}
[\![X]\!]_{\eta,\xi} &=& \xi(X) \\
[\![A \to B]\!]_{\eta,\xi} &=& [\![A]\!]_{\eta,\xi} \twoheadrightarrow [\![B]\!]_{\eta,\xi} \\
[\![\{l_1{:}A_1,\ldots,l_n{:}A_n\}]\!]_{\eta,\xi} &=& \{l_1 = [\![A_1]\!]_{\eta,\xi},\ldots,l_n = [\![A_n]\!]_{\eta,\xi}\} \\
[\![\mathbf{Sig}\ I]\!]_{\eta,\xi} &=& \mathsf{SIG}([\![I]\!]_{\eta,\xi}) \\
[\![\mathbf{Class}\ I\ \mathbf{with}\ init,m]\!]_{\eta,\xi} &=& \mathsf{CLASS}([\![I]\!]_{\eta,\xi}, [\![init]\!]_\eta, [\![m]\!]_\eta)
\end{array}$$

Again, the notation for record types is abused as shorthand for their interpretations under the encoding discussed in Remark 4.1.

Theorem 5.6
If $\Gamma \vdash a \simeq a' : A$ *in* λ^{OO}
then $([\![a]\!]_\eta, [\![a']\!]_\eta) \in [\![A]\!]_{\eta,\xi}$ *for all* $(\eta,\xi) \models \Gamma$.

Proof. Soundness of type assignment, i.e.

$$\Gamma \vdash a : A \ \wedge\ (\eta,\xi) \models \Gamma \Rightarrow [\![a]\!]_\eta \sqsubseteq [\![A]\!]_{\eta,\xi},$$

can be proved in the usual way. Lemma 5.3.1 is needed for soundness of the subtyping rule for classes, 5.3.2 for soundness of the introduction rule for classes, and 5.3.3 – together with Lemma 4.6 – for soundness of the elimination rules for classes.

No extra work is needed to prove soundness of reduction: we can reuse the following property of the PER-interpretation of $F_\leq$: if $[\![a]\!]_\eta$ and $[\![a']\!]_\eta$ are defined, then

$$a =_\beta a' \Rightarrow [\![a]\!]_\eta = [\![a']\!]_\eta.$$

Since the mapping from λ^{OO} to $F_\leq$ preserves reduction, we immediately have the property that if $[\![a]\!]_\eta$ and $[\![a']\!]_\eta$ are defined, then

$$a \simeq a' \Rightarrow [\![a]\!]_\eta = [\![a']\!]_\eta. \qquad \square$$

5.1 Subtyping is behavioural subtyping

We now show that in the PER model subtyping between class types corresponds with the notion of behavioral subtyping as we informally explained it in Section 2.

Definition 5.7 *For* $init \sqsubseteq Rep$ *and* $m \sqsubseteq Rep \twoheadrightarrow (Rep)$ *the per* $\mathsf{REACH}(\mathcal{I}, Rep, init, m)$ *is defined as follows:*

$$\begin{array}{l}
\mathsf{REACH}(\mathcal{I}, Rep, init, m) \\
= \\
\bigcap\{X \sqsubseteq Rep \mid init \sqsubseteq X \ \wedge\ m \sqsubseteq X \twoheadrightarrow \mathcal{I}(X)\}.
\end{array}$$

$\mathbb{N}/\mathsf{REACH}(\mathcal{I}, Rep, init, m)$ is the set of those Rep-equivalence classes reachable from the state *init* using the method implementations m. Note the similarity between the definition of REACH and the definition of CLASS. There is close relationship between the two:

Lemma 5.8 *Let* $\mathcal{I} : \mathsf{PER} \to \mathsf{PER}$. *Suppose that* $\mathcal{I}$ *is continuous – i.e.* $\mathcal{I}(\bigcap_i X_i) = \bigcap_i \mathcal{I}(X_i)$ *– and that* $\mathcal{I}(R);\mathcal{I}(S) \subseteq \mathcal{I}(R;S)$ *for all pers* R *and* S. *Then*

$$\begin{array}{l}
\mathbb{N}/\mathsf{CLASS}(\mathcal{I}, init, m) \\
= \\
\{[\mathsf{obj}\langle s,m\rangle]_{\mathsf{SIG}(\mathcal{I})} \mid s \sqsubseteq \mathsf{REACH}(\mathcal{I}, Rep, init, m)\}
\end{array}$$

Proof. (Sketch) First we consider ($\subseteq$). Define the per X as

$$X = Rep\ \cap\ (f;\mathsf{CLASS}(\mathcal{I}, init, m); f^{\leftarrow}),$$

where $f \subseteq \mathbb{N} \times \mathbb{N}$ is the relation $\{(s, \mathsf{obj}\langle s,m\rangle) \mid s \in \mathbb{N}\}$ and $f^{\leftarrow}$ its inverse. We can prove the following properties of X:

- $X \sqsubseteq Rep$,
- $init \sqsubseteq X$,

- $m \sqsubseteq X \twoheadrightarrow \mathcal{I}(X)$.

It then follows by the definition of REACH that

$$\mathsf{REACH}(\mathcal{I}, Rep, init, m) \subseteq X \ ,$$

and so

$$\begin{array}{ll} & f^{\leftarrow}; \mathsf{REACH}(\mathcal{I}, Rep, init, m); f \\ \subseteq & f^{\leftarrow}; X; f \\ = & f^{\leftarrow};\ (Rep \ \cap \ f; \mathsf{CLASS}(\mathcal{I}, init, m); f^{\leftarrow})\ ; f \\ \subseteq & (f^{\leftarrow}; Rep; f) \ \cap \ (f^{\leftarrow}; f; \mathsf{CLASS}(\mathcal{I}, init, m); f^{\leftarrow}; f) \\ = & (f^{\leftarrow}; Rep; f) \ \cap \ \mathsf{CLASS}(\mathcal{I}, init, m) \\ \subseteq & \mathsf{CLASS}(\mathcal{I}, init, m) \end{array}$$

and ($\subseteq$) follows directly from the inclusion above.

Now to prove ($\supseteq$). Define

$$Y = \mathsf{SIG}(\mathcal{I}); f^{\leftarrow}; \mathsf{REACH}(\mathcal{I}, Rep, init, m); f; \mathsf{SIG}(\mathcal{I})$$

For Y we can prove the following properties:

- $Y \sqsubseteq \mathsf{SIG}(\mathcal{I})$,
- $(init, m) \sqsubseteq Y$,
- $out_{\mathcal{I}} \sqsubseteq Y \twoheadrightarrow \mathcal{I}(Y)$.

It then follows by the definition of CLASS that

$$\mathsf{CLASS}(\mathcal{I}, init, m) \subseteq Y \ ,$$

from which we can prove ($\supseteq$). □

The relation below defines subtyping between interpretations of signatures:

Definition 5.9 *The relation $\leq$ on PER→PER is defined as follows:*

$$\begin{array}{ll} & I_1 \leq I_2 \\ \iff & \\ & \mathcal{I}_1(X) \subseteq \mathcal{I}_2(X) \textit{ for all pers } X \textit{ and} \\ & \textit{both } \mathcal{I}_i \textit{ are continuous with} \\ & \mathcal{I}_i(R); \mathcal{I}_i(S) \textit{ for all pers } R \textit{ and } S. \end{array}$$

We can now state our main result, namely that, for interpretations of class types, the subset relation on pers is equivalent with the notion of behavioural subtyping that we described in Section 2.

Theorem 5.10 (Subtyping is Behavioural Subtyping)

Suppose $init_i \sqsubseteq Rep_i$ and $m_i \sqsubseteq Rep_i \twoheadrightarrow \mathcal{I}_i(Rep_i)$, for certain pers Rep_i for $i = 1, 2$. If $\mathcal{I}_1 \leq \mathcal{I}_2$ then

$$\begin{array}{l} \mathsf{CLASS}(\mathcal{I}_1, init_1, m_1) \subseteq \mathsf{CLASS}(\mathcal{I}_2, init_2, m_2) \\ \iff \\ \exists \sim \subseteq \mathbb{N} \times \mathbb{N}. \ \ \sim = Rep_1; \sim; Rep_2 \ \wedge \\ \qquad (m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}_2(\sim) \ \wedge \\ \qquad \forall s_1 \sqsubseteq \mathsf{REACH}(\mathcal{I}_1, Rep_1, init_1, m_1). \\ \qquad\ \exists s_2 \sqsubseteq \mathsf{REACH}(\mathcal{I}_2, Rep_2, init_2, m_2).\ s_1 \sim s_2 \end{array}$$

The second part of this theorem is a formal definition of the notion of behavioural subtyping discussed in Section 2: The condition

$$(m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}_2(\sim)$$

corresponds to condition (i) on page 3, and the condition

$$\begin{array}{l} \forall s_1 \sqsubseteq \mathsf{REACH}(\mathcal{I}_1, Rep_1, init_1, m_1). \\ \ \exists s_2 \sqsubseteq \mathsf{REACH}(\mathcal{I}_2, Rep_2, init_2, m_2).\ s_1 \sim s_2 \end{array}$$

corresponds to condition (ii) on page 3.

Proof. (Sketch) Define $C_i = \mathsf{CLASS}(\mathcal{I}_i, init_i, m_i)$ and $R_i = \mathsf{REACH}(\mathcal{I}_i, Rep_i, init_i, m_i)$.

($\Rightarrow$) Let $C_1 \subseteq C_2$.

Define $\sim \subseteq \mathbb{N} \times \mathbb{N}$ as follows

$$\sim = Rep_1; f_1; \mathsf{SIG}(\mathcal{I}); f_2^{\leftarrow}; Rep_2,$$

where $f_i = \{(s, \mathsf{obj}\langle s, m_i\rangle) \mid s \in \mathbb{N}\}$. For this relation $\sim$ the required properties can be proven.

($\Leftarrow$) Let $\sim \subseteq \mathbb{N} \times \mathbb{N}$ be such that

(i) $\sim = Rep_1; \sim; Rep_2$,

(ii) $(m_1, m_2) \in \sim \twoheadrightarrow \mathcal{I}_2(\sim)$,

(iii) $\forall s_1 \sqsubseteq R_1.\ \exists s_2 \sqsubseteq R_2.\ s_1 \sim s_2$.

Suppose that $o_1 \sqsubseteq C_1$. Then by Lemma 5.8($\subseteq$) there is an $s_1 \sqsubseteq R_1$ such that $(o, (s_1, m_1)) \in \mathsf{SIG}(\mathcal{I}_1)$. Then by (iii) there is an $s_2 \sqsubseteq R_2$ such that $s_1 \sim s_2$. By Property 4.3 it now follows that $(\mathsf{obj}\langle s_1, m_1\rangle, \mathsf{obj}\langle s_2, m_2\rangle) \in \mathsf{SIG}(\mathcal{I}_2)$, and then $(s_2, m_2) \sqsubseteq C_2$ by Lemma 5.8($\supseteq$). Moreover, by $\mathsf{SIG}(\mathcal{I}_1) \subseteq \mathsf{SIG}(\mathcal{I}_2)$ and the transitivity of pers: $(o_1, \mathsf{obj}\langle s_2, m_2\rangle) \in \mathsf{SIG}(\mathcal{I}_2)$.

This proves

$$o_1 \sqsubseteq C_1 \ \Rightarrow\ \exists o_2 \sqsubseteq C_2.\, (o_1, o_2) \in \mathsf{SIG}(\mathcal{I}_2).$$

From this property we can now deduce $C_1 \subseteq C_2$ using $C_i \sqsubseteq \mathsf{SIG}(\mathcal{I}_i)$ and some basic properties of $\sqsubseteq$. □

6 Conclusions and directions for future work

This paper establishes a link between three different strands of research on object-oriented languages, namely

- the type-theoretic approach to objects of [PT94],
- the work on behavioural subtyping of [Lea90],
- the categorical approach to objects of [Rei95].

For an extension of the type-theoretic encoding of object of Pierce and Turner [PT94] we have shown that the standard interpretation of subtyping in PER models – subtypes are subpers – provides exactly the notion of behavioural subtyping defined by Leavens [Lea90]. The crucial property is that object types are interpreted as final co-algebras. The correspondence between the existential object encoding and final coalgebras noted in [HP95] extends to our class types and sub-coalgebras of the final coalgebra. Sub-coalgebras are used in [Rei95] and [Jac96] as specifications of objects; our class types can of course be regarded as specifications, where we specify objects by giving a particular implementation.

The usefulness of the coalgebraic view of objects suggests that it might be better to use a primitive notion of coinductive type to present the existential object model, rather than an encoding of such types using existential types. The existential object model could for instance be carried out using Hagino's categorical datatypes [Hag87] extended with subtyping. (The interface types of λ^{OO} are essentially coalgebraic types in the sense of [Hag87].) An advantage would be that coinductive types only require a first-order type system, whereas existential types require a second-order type system.

One subject for future work is a more general description of a model for λ^{OO} in categorical terms, in which interface types are interpreted as final coalgebras, class types as subcoalgebras, and subtyping as coercions between them. We hope this will streamline much of the theory, and allow a presentation giving more than just sketches of proofs. (Note that I-coalgebras are only defined up to isomorphism, but the PER model here relies on the construction of a particular one of these as the interpretation of an interface type.)

We have not mentioned inheritance here. Inheritance for the existential model encoding is described in [PT94]. Now that we have a notion of behavioural subtyping, the interesting problem to look at is: "When does inheritance produce behavioural subtypes ?". Ideally we would want to formulate general conditions that are sufficient to guarantee that a class defined by inheritance is a behavioural subtype of the class it inherits from.

λ^{OO} could be extended with subtyping between class types, where this subtyping between class types is *declared* by the programmer. It would have to be the responsibility of the programmer that such declared subtyping is sound, as this is not something that can be decided by a typechecker. We would then really want a logic for reasoning about programs in which soundness of subtyping between class types can be expressed and (dis)proved. Such a logic would be an major topic for further investigation. Here it might be possible to use existing work on behavioural subtyping.

Finally, it would be interesting to see if the PER models of the other object encodings, e.g. those discussed in [BCP97], can also provide a notion of behavioural subtyping for class types. This would be more difficult: these other object encodings are in type systems with unrestricted recursion, and it is not clear what the effect of recursion would be. Also, the method updates allowed by some of these encodings would cause complications. These would have to be ruled out if we want to statically guarantee that all objects of a class type have the same method table.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 682–696. Springer-Verlag, 1997.

[Ame89] Pierre America. A Behavioural Approach to Subtyping in Object-Oriented Languages. Technical Report Technical Report 443, Philips Research Laboratories, 1989.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS'97), Sendai, Japan*, volume 1281, pages 415–438. Springer LNCS, September 1997.

[BFAS90] E.S. Bainbridge, P.J. Freyd, A.Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990.

[BL90] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in [GM94].

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, 1992. Also in [GM94].

[FM94] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software (TACS'94) , Sendai, Japan*, volume 789 of *LNCS*, pages 844–886. Springer, 1994.

[FM97] K. Fisher and J.C. Mitchell. On the relationship between classes, objects, and data abstraction. In *Proceedings of the International Summer School on Mathematics of Program*, LNCS. Springer, 1997.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A Poigné, and D.E. Rydeheard, editors, *Category and Computer Science*, pages 140–157. Springer, September 1987.

[Has94] Tyu Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4:71–109, 1994.

[HP95] Martin Hofmann and Benjamin C. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

[Jac96] Bart Jacobs. Objects and classes, coalgebraically. In *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996. ISBN 0-7923-9770-3.

[Lea90] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990.

[Lis88] Barbara H. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(3), 1988.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mau95] Ian Maung. On simulation, subtyping and substitutability in sequential object systems. *Formal Aspects of Computing*, 7(6):620–651, 1995.

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Lang. and Syst.*, 10(3):470–502, 1988.

[PA93] Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375, 1993.

[PAC94] Gordon Plotkin, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 310–319, 1994.

[PH97] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Technische Universität München, 1997.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Rei95] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.