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 subtypes — 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 $F^*_{EC}$, but the class types we want to consider are not encodable in $F^*_{EC}$. Therefore we will define a simple object-oriented functional language $\lambda^{OO}$, that is essentially just some syntactic sugar for the existential object encoding, but extended with a notion of class type.

The standard model of $F^*_{EC}$ 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
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 Lisov’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] [Man95].

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] [PL97] [ALO97]. This second approach can combined with the first in order to cope with pre- and post-conditions that refer to the abstract values of objects. For example, suppose objects of type $ListA$ are specified in terms of the sequences in $A$ as they represent, and objects of type $SetA$ in terms of the sets in $P(A)$ they represent. Then a (simulation) relation $R \subseteq SetA \times P(A)$ 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 $\lambda^{OO}$. Section 4 discusses the PER model for $F_{\leq}$ of [CAC94], which provides a PER model for $\lambda^{OO}$ without class types. This PER model is then used in Section 5 as the basis of a PER model for $\lambda^{OO}$ 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 $\lambda^{OO}$

We give some simple examples to introduce $\lambda^{OO}$ 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

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

is the type of objects with methods $\text{getcount}$ and $\text{count}$, where $\text{getcount}$ returns a natural number and $\text{count}$ 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 $o \leftarrow \text{getcount} : \text{Nat}$ and $o \leftarrow \text{count} : \text{Counter}$.

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

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

This object is written as object $(s, m)$.

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

$m \ = \ \lambda s : \{x : \text{Nat}\}. \{\text{get} = s.x, \text{count} = \{x = s.x + 1\} \}$

: $(x : \text{Nat}) \rightarrow CounterI(\{x : \text{Nat}\})$

E.g. object $(\{x = 5\}, m)$ 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

$$(\text{object } (s, m)) \leftarrow \text{getcount} \ = \ (m.s) \text{getcount}$$

$$(\text{object } (s, m)) \leftarrow \text{count} \ = \ \text{object } ((m.s).\text{count}, m)$$

For example, if $o \equiv \text{object } (\{x = 5\}, m)$, then $o \leftarrow \text{getcount} = 5$ and $o \leftarrow \text{count} = \text{object } (\{x = 6\}, m)$. Note that the methods $\text{count}$ and $\text{getcount}$ are treated differently, because $\text{count}$ returns an object of the same class and $\text{getcount}$ just returns a natural number. We will call a method such as $\text{count}$ a mutator and a method such as $\text{getcount}$ 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 $\text{Sig}

CounterI$ for

$$\exists Rep. Rep \times (Rep \rightarrow \text{CounterI}(Rep))$$

and object $(s, m)$ for

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

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 $\text{object } (s, m)$ can only be observed indirectly by invoking the methods $\text{getcount}$ and $\text{count}$.

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: $\text{object } (s_1, m_1)$ and $\text{object } (s_2, m_2)$ of type $\text{Counter}$ 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 \{(m_1.x_1).\text{getcount} =_{\text{Nat}} (m_2.x_2).\text{getcount} \}

\{m_1.x_1).\text{count} \sim (m_2.x_2).\text{count}\}$$

\text{counter} = \text{object } (s, m)$.
2.2 Subtyping on Interface Types

An example of a signature subtype of Counter is

\[ R\text{Counter} = \text{Sig} \text{RCounterI}, \]

where

\[ R\text{CounterI} \left( X \right) = \{ \text{getcount} : \text{Nat}, \text{count} : X, \text{reset} : X \}. \]

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

The PER model does indeed interpret RCounter and Counter as subsets: \([\text{RCounter}] \subseteq [\text{Counter}]\). We can think of the partial equivalence relations \([\text{RCounter}]\) and \([\text{Counter}]\) as the notion of equality for counters and resettable counters, respectively. Then \([\text{RCounter}] \subseteq [\text{Counter}]\) means that \([\text{Counter}]\) is a coarser notion of equality than \([\text{RCounter}]\). 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 \([\text{Counter}]\), but would not be identified by \([\text{RCounter}]\).

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

\[ \text{CounterClass} = (\text{Class RCounterI with init, m}) \]

where

\[
\begin{align*}
\text{init} & = \{ x = 1 \} \\
\text{m} & = \lambda s : \text{Nat}. \{ \text{get} = s.s, \text{count} = \{ x = s.s + 1 \} \}
\end{align*}
\]

Intuitively, CounterClass is the following inductively defined set:

- \text{object} (\text{init}, m) : \text{CounterClass},
- \text{if} \ c : \text{CounterClass} \text{ then} \text{c} -\text{count} : \text{CounterClass}.

i.e. CounterClass can be understood as the smallest subset of Counter that contains \text{object} (\text{init}, m) 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 \text{object} (\text{init}, m).

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

\[
\forall o : \text{CounterClass}.
\]

Interpreting class types in the PER model is not a problem, and we get the expected relation between the pers interpreting \text{CounterClass} and \text{Counter}:

\[
[\text{CounterClass}] \subseteq [\text{Counter}].
\]

The pers \([\text{CounterClass}]\) and \([\text{Counter}]\) provide the same notion of equality, i.e. all \([\text{CounterClass}]\) equivalence classes are also \([\text{Counter}]\) equivalence classes, but there are fewer \([\text{CounterClass}]\) equivalence classes than there are \([\text{Counter}]\) equivalence classes.

2.4 Subtyping on Class Types

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

\[ \text{RCounterClass} = (\text{Class RCounterI with init}_R, m_R) \]

where

\[
\begin{align*}
\text{init}_R & = \{ y = \text{cons}0 \{ \text{cons}0 \text{nil} \} \} \\
\text{m}_R & = \lambda s : \text{natlist}. \{ \text{getcount} = \text{length}(y), \text{count} = \{ y = \text{cons}0 \text{y} \}, \text{reset} = \{ y = \text{cons}2 \{ \text{cons}2 \text{nil} \} \} \}
\end{align*}
\]

RCounterClass can be understood as the smallest subset of RCounter that contains \text{object} (\text{init}_R, m_R) and is closed under method invocation.

RCounterClass can be viewed as a behavioural sub-type 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 \text{o} : RCounterClass there is a \text{o} : CounterClass that is indistinguishable from \text{o}. More formally, we can say that RCounterClass is a behavioural subtype of CounterClass because there exists a relation \sim \subseteq \{ x : \text{Nat} \} \times \{ y : \text{natlist} \} such that

\[
\begin{align*}
\text{s} \sim \text{s}' & \Rightarrow \{ (m \text{s}).\text{getcount} = \text{Nat} (m \text{R} \text{s}').\text{getcount} \\
& \quad (m \text{s}).\text{count} \sim (m \text{R} \text{s}').\text{count} \} & (i)
\end{align*}
\]

and

\[
\forall o : \text{RCounterClass}.
\]

\[
\exists o : \text{CounterClass}.
\]

\[
\text{o} -\text{count} : \text{CounterClass}.
\]

the states of \text{o} and \text{o}' are related by \sim.

This relation \sim is of course

\[
\text{s} \sim \text{s}' \Rightarrow \text{s} = \text{length}(\text{s}, \text{y}).
\]

Together conditions (i) and (ii) guarantee that for every \text{o} : RCounterClass there is a \text{o}' : CounterClass such that \text{o}' is indistinguishable from \text{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 \text{o} -\text{getcount} \geq 2 for all \text{o} : RCounterClass.

Examples of classes that are not behavioural subtypes of CounterClass are

\[
\begin{align*}
\text{CounterClass}_{\text{1}} & = (\text{Class CounterI with init, m'}) \\
\text{RCounterClass}_{\text{1}} & = (\text{Class CounterI with init}_R, m_R)
\end{align*}
\]
with

\[
m'_R = \lambda x. \{ \text{get} = s.x. \text{count} = \{ x = s.x + 2 \} \}
\]

These classes are not behaviourally subtypes of \textit{CounterClass}, for slightly different reasons. For \textit{CounterClass1} we cannot find a suitable simulation relation. For \textit{RCounterClass1} there is an obvious simulation relation, namely the same one we used for \textit{RCounterClass}. However, there is an object in \textit{RCounterClass1} for which there is no related object in \textit{CounterClass}: resetting a counter in \textit{RCounterClass1} produces an object \( o \equiv \text{object} \{ \{ y = \text{null} \}, m \} \) for which \( o \rightarrow \text{get count} = 0 \), and this object cannot be related to any object in \textit{CounterClass}, since all counters in \textit{CounterClass} have a get count greater than 0.

For class types, the interpretation of subtyping in the PER model – subtypes are subpro – 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

\[
\llbracket \text{CounterClass} \rrbracket \not\subseteq \llbracket \text{CounterClass1} \rrbracket \not\subseteq \llbracket \text{CounterClass} \rrbracket
\]

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

3 Definition of \( \lambda^{\text{OO}} \)

The raw syntax of the terms \( a \), types \( A \) and signatures \( I \) of \( \lambda^{\text{OO}} \) is given by the following grammar

\[
a ::= x | \lambda x: A \ a | a \ a | \{ l_1 = a, \ldots, l_n = a \} \ a, l | \ a \ (a) | \ a \ \text{object} \ (A, a, a) \\
A ::= X | A \rightarrow A' | \{ l_1: A_1, \ldots, l_n: A_n \} \\
\text{Sig} I \ | \ \text{Class} \ / \ \text{with} \ A, a, a \\
I ::= (X) \{ l_1: A_1 \rightarrow A', \ldots, l_n: A_n \rightarrow A' \}
\]

Here \( x \) ranges over term variables, \( X \) ranges over type variables, and \( I \) over a countable set \( L \) 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 (\textit{type}) or interface.

For simplicity, we divide \( L \) into a set \( L_{\text{obs}} \) of labels that can be used as names for \textit{observer methods} and a set \( L_{\text{mut}} \) of labels that can be used as names for \textit{mutator methods}, and we insist that

\[
l_i \in L_{\text{obs}} \Rightarrow X \not\in \text{FV}(A_i \rightarrow B_i) \\
l_i \in L_{\text{mut}} \Rightarrow X \not\in \text{FV}(A_i) \land X \equiv B_i
\]

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

The contexts of \( \lambda^{\text{OO}} \) are given by

\[
\Gamma ::= e | \Gamma, x : A
\]

with no variable occurring twice.

The subtyping and typing rules of \( \lambda^{\text{OO}} \) 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 \( \Gamma \) if \( A \) and \( B \) are well-formed in \( \Gamma \), etc.)

The reduction relation \( \triangleright \) on terms is given by the rules

\[
\begin{align*}
& (x : A. b) \ d \triangleright [a / x]b \\
& \{ l_1 = a_1, \ldots, l_n = a_n \} h \triangleright a_i \\
& \text{object} \ (s, m) \leftarrow l \ (a) \ (m) \ l \ a \ (m) : l \ a \ (m) \\
& \text{object} \ (s, m) \leftarrow l \ (a) \ (m) \ l \ a \ (m) \ (m) : l \ a \ (m)
\end{align*}
\]

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 \( \triangleright \), and \( \Gamma \vdash a \simeq a' : A \) for \( \Gamma \vdash a : A \land \Gamma \vdash a' : A \land a \simeq a' \).

4 PER Model for \( \lambda^{\text{OO}} \) without Class Types

As we mentioned earlier, with the exception of the class types, \( \lambda^{\text{OO}} \) just introduces some syntactical sugar for the existential object encoding of objects in \( F_{\leq} \) given in [PT91]. In fact, we only need the subsystem \( F_{\leq} \) of \( F_{\leq}^\omega \) as target system.\footnote{In [PT91] \( F_{\leq}^\omega \) 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 \( " \leftarrow \" \). We don’t have these in \( \lambda^{\text{OO}} \).} So any model of \( F_{\leq} \) can be used as model for \( \lambda^{\text{OO}} \) without classes.

The model we use is the PER model of [PAC94]. This model combines the PER models of [BL90] and [BFA90]; it is essentially just the model of [BL90] – and interprets subtypes as subpro – but it uses the interpretation of polymorphic types given in [BFA90] – 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 \( \lambda^{\text{OO}} \) 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:

\[
\text{Sig} I = \exists X. X \times (X \rightarrow I(X))
\]

Remark 4.1 Here \( \exists X \) and \( \exists I \) stand for the usual \( F_{\leq} \) encodings. The record types we have in \( \lambda^{\text{OO}} \) 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 \( B_1 \times \ldots \times B_m \) where \( m \) is the greatest index of the \( l_i \) and \( B_j = A_j \) if \( j \) is the index of \( l_i \) and \( B = T o p \) 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.
The \(\lambda^{O^O}\) syntax for object formation and method invocation can be interpreted by the following \(F_\leq\)-terms:

\[
\begin{align*}
\text{object}_\ell (s, m) &= \text{pack} (\text{Rep}_\ell (s, m)) \to \text{Sig I} \\
\text{a} \text{++}-l(a) &= \text{open} \ a \text{ as } (X, (s, m)) \text{ in } \{m \mid a\} \text{ l} \\
\end{align*}
\]

Note that some type information is missing in the \(\lambda^{O^O}\)-terms, namely the type \(\text{Rep}\) of \(s\) in the first clause and \(\text{Sig I}\) 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 \(N\), and terms are interpreted as natural numbers, using some enumeration of the partial recursive functions.

The \(\text{per}\) interpreting a type \(A\) is written as \(\lfloor A \rfloor_\xi\), where \(\xi\) is a type environment that maps from type variables to pers. The natural number interpreting a term \(a\) is written as \(\lfloor a \rfloor_\eta\), where \(\eta\) is a term environment that maps to natural numbers. Another way of looking at the PER model is to interpret a type \(A\) as the set of \(\lfloor A \rfloor_\xi\)-equivalence classes, and to interpret a term \(a : A\) as the \(\lfloor A \rfloor_\xi\)-equivalence class that contains \(\lfloor a \rfloor_\eta\).

\(\lambda^{O^O}\) 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:

\[
\begin{align*}
\lfloor \text{Sig I} \rfloor_\xi &= \{\exists X. X \times (X \to I(X))\} \\
\lfloor \text{object}_\ell (s, m) \rfloor_\eta &= \{\text{pack} (\text{Rep}_\ell (s, m)) \to \text{Sig I} \} \\
\lfloor \text{o} \text{++}-l(a) \rfloor_\eta &= \{\text{open} \ a \text{ as } (X, (s, m)) \text{ in } \{m \mid a\} \text{ l} \} \\
\end{align*}
\]

A pair \((\eta, \xi)\) satisfies a context \(\Gamma\) written \(\eta(\xi) \models \Gamma\) iff \((\eta(x), \eta(x)) \in \lfloor A \rfloor_\eta\) for all declarations \(x : A\) in \(\Gamma\). Given that the PER model is a sound model for \(F_\leq\), it is also a sound model for \(\lambda^{O^O}\).

**Theorem 4.2**

If \(\Gamma \vdash a \simeq a' : A\) in \(\lambda^{O^O}\) without class types, then \((\lfloor a \rfloor_\eta, \lfloor a' \rfloor_\eta) \in \lfloor A \rfloor_\eta\) for all \((\eta, \xi)\) satisfying \(\Gamma\).

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 \(n \vdash m\) for the \(n^{th}\) partial recursive function applied to \(m\), and \(\lambda x. E(x)\) for the index of a partial recursive function for which \(\lambda x. E(x)\n = E(n)\), where \(E(x)\) is a partial recursive description of a natural number depending on \(x\).

We write \(\lfloor n \rfloor_R\) for the \(R\)-equivalence containing \(n\), and \(\lfloor N \rfloor_R\) for the set of \(R\)-equivalence classes. For pers \(R\) and \(S\), the per \(R \Rightarrow S\) is defined as

\[
\begin{align*}
R \Rightarrow S &= \{(f, f') \in \lfloor N \times N \mid \forall (a, a') \in A. (f \cdot a, f' \cdot a') \in B)\}.
\end{align*}
\]
We write \( n \in R \) as abbreviation for \((n,n) \in R\).

The category \( \text{PER} \) is defined as in [BFA890] 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 \( R \Rightarrow S \).

### 4.1 Interpretation of objects

We define \( \text{obj}(\cdot) \) and \( \cdot \iff \cdot \) as the interpretations of object \( (\cdot,\cdot) \) and \( \cdot \iff \cdot \) in the PER model. So

\[
\text{obj}_I(\langle s,m \rangle)_{\eta} = \text{obj}(\langle [s], [m] \rangle_{\eta})
\]

and

\[
[\cdot \iff \cdot]_{\eta} = [\cdot]_{\eta} \iff [\cdot]_{\eta}.
\]

Looking at the interpretations of the \( F \) terms, we find that

\[
\text{obj}(s,m) \iff a = \begin{cases}
(m \cdot s) \cdot l \cdot a & \text{if } l \in \mathcal{L}_{\text{out}} \\
\text{obj}(m \cdot s) \cdot l \cdot a \cdot m & \text{if } l \in \mathcal{L}_{\text{in}}
\end{cases}
\]

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

### 4.2 Interpretation of interface types

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

We define \( \text{Sig}(\cdot) \) as the interpretation of \( \text{Sig}(\cdot) \) in the \( \text{PER} \) model. So

\[
\text{Sig}_I(\cdot)_{\xi} = \exists X. X \times (X \to I(X))_{\xi} = \text{Sig}(\langle \cdot \rangle_{\xi})
\]

Let \( I : \text{PER} \Rightarrow \text{PER} \) be the functor interpreting some interface \( I \) in some environment \( \iota \), i.e. \( I = [\cdot]_{\iota} \). We define \( \text{out}_X \) as

\[
\text{out}_X = \lambda \omega. (\iota s). \text{obj}(s, \text{snd}(\omega)) \cdot (\text{snd}(\omega) \cdot \text{fst}(\omega)).
\]

Another way of defining \( \text{out}_X \) is as the interpretation of \( \text{out}_I : \text{Sig} I \Rightarrow I(\text{Sig} I) \), defined as follows

\[
\text{out}_I = \lambda o. \text{Sig} I. \text{open} as (X, (s, m)) \text{ in } I_m(\lambda x : X. \text{obj}_I(\langle x, m \rangle)) \text{ (m s)} : \text{Sig} I \Rightarrow I(\text{Sig} I)
\]

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 \( \langle \text{Sig}(I), \text{out}_X \rangle \) is a final coalgebra \[\text{Has94]} | P A93]. So

\[
\text{out}_X \in \text{Sig}(I) \Rightarrow I(\text{Sig}(I)),
\]

and we have the following properties:

**Property 4.3** Let \( s_1 \in \text{Rep}_I \) and \( m_2 \in \text{Rep}_I \Rightarrow I(\text{Rep}_I) \) for certain pers \( \text{Rep}_I \), \( i = 1, 2. \)

Then

\[
(\text{obj}(s_1, m_1) \text{obj}(s_2, m_2)) \in \text{Sig}(I) \equiv \exists \sim \in X \times X. \sim = R_{\text{Rep}_I; \sim} \land (s_1, m_2) \in \sim \Rightarrow I(\sim)
\]

Here \( ; \) denotes composition of relations.

**Property 4.4** For any relation \( \sim \subseteq X \times X \)

\[
\text{out}_X \subseteq \sim \Rightarrow I(\sim) \Rightarrow \sim \subseteq \text{Sig}(I).
\]

In other words, \( \text{Sig}(I) \) is the maximum bisimulation.

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

**Corollary 4.5** Let \( (s_1, s_2) \in \text{Rep} \) and \( (m_1, m_2) \in \text{Rep} \Rightarrow I(\text{Rep}) \) for some pers \( \text{Rep} \).

Then \( \langle \text{obj}(s_1, m_1), \text{obj}(s_2, m_2) \rangle \in \text{Sig}(I) \).

The mapping \( \text{out}_X \) is related to the interpretation of method invocations \( \cdot \iff \cdot \) as follows:

**Property 4.6** Let \( I = [\cdot]_{\xi} \) for some \( \xi \), with \( I \) an interface that contains a method \( l \). Then

\[
\text{out}_X = \lambda l. \text{Sig} I. \text{open} as (X, (s, m)) \text{ in } I_m(\lambda x : X. \text{obj}_I(\langle x, m \rangle)) (m s) : \text{Sig} I \Rightarrow I(\text{Sig} I)
\]

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

### 5 Model for \( \lambda^{OO} \) with Class Types

We now extend the interpretation of \( \lambda^{OO} \) without class types in the \( \text{PER} \) model of [PA94] to an interpretation of the full \( \lambda^{OO} \) including the class types.

**Definition 5.1** The relation \( \subseteq \) on pers is defined by

\[
R \subseteq S \iff \mathbb{N}/R \subseteq \mathbb{N}/S.
\]

An equivalent definition is

\[
R \subseteq S \iff R \subseteq S \land S \subseteq R \land S \cdot R = S \cdot S.
\]

The relation \( \subseteq \) is used to define the interpretation of a class types:

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

\[
\text{CLASS}(I, s, m) = \bigcap \{ X \subseteq \text{Sig}(I) \mid \text{obj}(s,m) \subseteq X \land \text{out}_X \subseteq X \Rightarrow I(X) \}
\]

This defines \( \mathbb{N}/\text{CLASS}(I, s, m) \) as the smallest subset of \( \mathbb{N}/\text{Sig}(I) \) that contains \( \text{obj}(s,m) \) and is closed under method invocations.

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

1. \( \text{CLASS}(I, s, m) \subseteq \text{Sig}(I) \)
2. \( \text{obj}(s, m) \in \text{CLASS}(I, s, m) \)
3. \( \text{out}_{I} \in \text{CLASS}(I, s, m) \rightarrow I(\text{CLASS}(I, s, m)) \)
4. Let \( (s_{1}, s_{2}) \in \text{Rep} \) and \( (m_{1}, m_{2}) \in \text{Rep} \rightarrow I(\text{Rep}). \)
   Then \( \text{CLASS}(I, s_{1}, m_{1}) = \text{CLASS}(I, s_{2}, m_{2}). \)

Proof. These properties easily follow from the definition of \( \text{CLASS} \) and the assumptions on \( I \). For 4. use Corollary 4.5 to deduce that
\[
(s, m) \in X \iff (s', m') \in X \quad \text{for any } X \subseteq \text{SIG}(I)
\]
from the assumptions on \( s_{1} \) and \( m_{1} \).

It is easy to verify that any \( I \) that is the interpretation of a \( \lambda^{\text{OO}} \)-signature will satisfy the conditions of the lemma above.

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

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

**Definition 5.4** The interpretation \( [a]_{\eta} \in \text{IN} \) of a \( \lambda^{\text{OO}} \)-term \( M \) in term environment \( \eta \) is defined as
\[
[a]_{\eta} = \text{Emse}(a)_{\eta},
\]
where
\[
\text{Erase} \left( \text{object}_{I}(s, m) \right) = \text{object}_{I}(\text{Emse}(s), \text{Emse}(m))
\]
\[
\text{Erase}(\text{let}_{I}(a)) = \text{Emse}(a)_{\eta} \subseteq (\text{Emse}(a))_{\eta}
\]
\[
\text{Emse}(\lambda x. b) = \text{Emse}(b)_{\eta}
\]
\[
\text{Emse}(f(a)) = \text{Emse}(f)_{\eta}(\text{Emse}(a))
\]
\[
\text{Emse}(a.l) = \text{Emse}(a)_{\eta} l
\]
\[
\text{Erase}(\{l_{1} = a_{1}, \ldots, l_{n} = a_{n}\})_{\eta} = \{l_{1} = \text{Emse}(a_{1})_{\eta}, \ldots, l_{n} = \text{Emse}(a_{n})_{\eta}\}
\]
and he interpretation of emsed terms is defined by
\[
\{f\}_{\eta} = \eta(x)
\]
\[
[x. \theta]_{\eta} = \lambda n. [\theta]_{\eta[n]}[\eta]
\]
\[
[\theta]_{\eta} = [\eta]_{\eta} \cdot [\eta]_{\eta}
\]
\[
[\text{object}_{I}(s, m)_{\eta}] = [\text{obj}_{I}(s), [m]_{\eta}]_{\eta}
\]
\[
[\text{let}_{I}(a)_{\eta}] = [a]_{\eta} \leftarrow [\eta]_{\eta}
\]
\[
\{l_{1} = a_{1}, \ldots, l_{n} = a_{n}\}_{\eta} = \{l_{1} = [a_{1}]_{\eta}, \ldots, l_{n} = [a_{n}]_{\eta}\}
\]
\[
[a]_{\eta} l = [a]_{\eta} l
\]

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 \( I \models a \equiv a' : A \) in \( \lambda^{\text{OO}} \),

then \( ([a]_{\eta}, [a']_{\eta}) \in \{A\}_{\eta, \xi} \) for all \( \eta, \xi \models I. \)

Proof. Soundness of type assignment, i.e.
\[
I \models a : A \wedge (\eta, \xi) \models I \Rightarrow [a]_{\eta} \subseteq \{A\}_{\eta, \xi},
\]
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 \( F_{\leq} : [a]_{\eta} \leq [a']_{\eta} \),

\[
[a]_{\eta} \leq [a']_{\eta}.
\]

Since the mapping from \( \lambda^{\text{OO}} \) to \( F_{\leq} \) preserves reduction, we immediately have the property that if \( [a]_{\eta} \) and \( [a']_{\eta} \) are defined, then
\[
a \equiv a' \Rightarrow [a]_{\eta} = [a']_{\eta}.
\]

**5.1 Subtyping is behavioural subtyping**

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

**Definition 5.7** For \( \text{init} \in \text{Rep} \) and \( m \in \text{Rep} \rightarrow (\text{Rep}) \) the per \( \text{REACH}(I, \text{Rep}, \text{init}, m) \) is defined as follows:
\[
\text{REACH}(I, \text{Rep}, \text{init}, m) = \{ \forall X \subseteq \text{IN} \} \wedge \{ \forall X \subseteq \text{IN} \}
\]

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

**Lemma 5.8** Let \( I : \text{PER} \rightarrow \text{PER} \). Suppose that \( I \) is continuous – i.e. \( I(\bigcap X_{i}) = \bigcap I(\bigcap X_{i}) \) and that \( I(\bigcap R ; I(S)) \subseteq I(\bigcap R ; I(S)) \) for all \( \text{Rep} \) and \( S \). Then
\[
\text{CLASS}(I, \text{init}, m) = \{ [\text{obj}(s, m)]_{\text{SIG}}(s) | s \in \text{REACH}(I, \text{Rep}, \text{init}, m) \}
\]

Proof. (Sketch) First we consider \( \subseteq \). Define the per \( X \) as
\[
X = \text{Rep} \cap \{ f \subseteq \text{CLASS}(I, \text{init}, m) ; f^{+} \}
\]
where \( f \subseteq \text{IN} \times \text{IN} \) is the relation \( \{ (s, \text{obj}(s, m)) | s \in \text{IN} \} \) and \( f^{+} \) its inverse. We can prove the following properties of \( X \):

- \( X \subseteq \text{Rep} \)
- \( \text{init} \in X \)


• \( m \in X \rightarrow I(X) \).

It then follows by the definition of \( \text{REACH} \) that
\[
\text{REACH}(I, \text{Rep}, \text{init}, m) \subseteq X,
\]
and so
\[
f^* \cdot \text{REACH}(I, \text{Rep}, \text{init}, m); f
\subseteq f^* \cdot (\text{Rep} \cap f; \text{CLASS}(I, \text{init}, m); f^*); f
\subseteq (f^*; \text{Rep}; f) \cap \text{CLASS}(I, \text{init}, m); f^*; f
\subseteq \text{CLASS}(I, \text{init}, m)
\]
and (\( \subseteq \)) follows directly from the inclusion above.

Now we prove (2). Define
\[
Y = \text{SIG}(I); f^* \cdot \text{REACH}(I, \text{Rep}, \text{init}, m); f; \text{SIG}(I)
\]
For \( Y \) we can prove the following properties:
- \( Y \subseteq \text{SIG}(I) \),
- \( (\text{init},m) \in Y \),
- \( \text{out}_z \in Y \rightarrow I(Y) \).

It then follows by the definition of \( \text{CLASS} \) that
\[
\text{CLASS}(I, \text{init}, m) \subseteq Y,
\]
from which we can prove (2).

\( \square \)

The relation below defines subtyping between interpretations of signatures:

**Definition 5.9** The relation \( \leq \) on \( \text{PER} \rightarrow \text{PER} \) is defined as follows:
\[
I_1 \leq I_2 \iff 
I_1(X) \subseteq I_2(X) \text{ for all pers } X \text{ and both } I_i \text{ are continuous with } I_i(R); I_i(S) \text{ for all pers } R \text{ and } S.
\]

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 \( \text{init}_i \in \text{Rep}_i \) and \( m_i \in \text{Rep}_i \rightarrow I_i(\text{Rep}_i) \), for certain pers \( \text{Rep}_i \) for \( i = 1,2 \). If \( I_1 \leq I_2 \) then
\[
\text{CLASS}(I_1, \text{init}_1, m_1) \subseteq \text{CLASS}(I_2, \text{init}_2, m_2) \iff
\exists \sim \subseteq \mathbb{N} \times \mathbb{N}. \sim = \text{Rep}_1; \sim; \text{Rep}_2 \wedge
(m_1, m_2) \in \sim \rightarrow I_2(\sim) \wedge
\forall s_1 \in \text{REACH}(I_1, \text{Rep}_1, \text{init}_1, m_1). \exists s_2 \in \text{REACH}(I_2, \text{Rep}_2, \text{init}_2, m_2). s_1 \sim s_2
\]

The second part of this theorem is a formal definition of the notion of behavioural subtyping discussed in Section 2: The condition
\[
(m_1, m_2) \in \sim \rightarrow I_2(\sim)
\]
corresponds to condition (i) on page 3, and the condition
\[
\forall s_1 \in \text{REACH}(I_1, \text{Rep}_1, \text{init}_1, m_1). \exists s_2 \in \text{REACH}(I_2, \text{Rep}_2, \text{init}_2, m_2). s_1 \sim s_2
\]
corresponds to condition (ii) on page 3.

**Proof.** (Sketch) Define \( C_i = \text{CLASS}(I_i, \text{init}_i, m_i) \) and \( R_i = \text{REACH}(I_i, \text{Rep}_i, \text{init}_i, m_i) \).

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

Define \( \sim \subseteq \mathbb{N} \times \mathbb{N} \) as follows
\[
\sim = \text{Rep}_1; f_1; \text{SIG}(I_1); f_2^*; \text{Rep}_2,
\]
where \( f_i = \{(s, \text{obj}(s, m_i)) \mid s \in \mathbb{N} \} \).

For this relation \( \sim \) the required properties can be proven.

\( \Leftarrow \) Let \( \sim \subseteq \mathbb{N} \times \mathbb{N} \) be such that
\[
(i) \quad \sim = \text{Rep}_1; \sim; \text{Rep}_2,
(ii) (m_1, m_2) \in \sim \rightarrow I_2(\sim),
(iii) \forall s_1 \in R_1. \exists s_2 \in R_2. s_1 \sim s_2.
\]

Suppose that \( o_1 \in C_1 \). Then by Lemma 5.8(\( \subseteq \)) there is an \( s_1 \in R_1 \) such that \((o_1, (s_1, m_1)) \in \text{SIG}(I_1) \).

Then by (iii) there is an \( s_2 \in R_2 \) such that \( s_1 \sim s_2 \). By Property 4.3 it now follows that \((\text{obj}(s_1, m_1), \text{obj}(s_2, m_2)) \in \text{SIG}(I_2) \)
and then \((s_2, m_2) \in C_2 \) by Lemma 5.8(\( \subseteq \)).
Moreover, by \( \text{SIG}(I_1) \subseteq \text{SIG}(I_2) \) and the transitivity of pers \((o_1, \text{obj}(s_2, m_2)) \in \text{SIG}(I_2) \).

This proves
\[
o_1 \in C_1 \Rightarrow \exists o_2 \in C_2. (o_1, o_2) \in \text{SIG}(I_2).
\]

From this property we can now deduce \( C_1 \subseteq C_2 \) using \( C_i \subseteq \text{SIG}(I_i) \) and some basic properties of \( \subseteq \).

\( \square \)

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 \( \Lambda^{OO} \) 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 \( \Lambda^{OO} \) in categorical terms, in which interface
types are interpreted as final coalgebras, class types as sub-
calgebras, 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 J-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.

$\lambda^{oo}$ 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 pro-
grams in which soundness of subtyping between class types
is 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 subtyp-
ing for class types. This would be more difficult: these other
object encodings are in type systems with unrestricted re-
cursion, and it is not clear what the effect of recursion would
be. Also, the method updates allowed by some of these en-
codings 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] Martin Abadi and Luca Cardelli. A Theory of Ob-
jects. Monographs in Computer Science. Springer,
1996.

of object-oriented programs. In TAPSOFT ’97: The-
ory and Practice of Software Development, pages

[Ame89] Pierre America. A Behavioural Approach to Sub-
typing in Object-Oriented Languages. Technical
Report Technical Report 443, Philips Research Labor-
atories, 1989.

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

[BFA90] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J.
Scott. Functorial polymorphism. Theoretical

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

programming language: Design, static typing and
semantics. Journal of Functional Programming,

[Car88] Luca Cardelli. A semantics of multiple inher-
itance. Information and Computation, 76:138–

[Car92] Luca Cardelli. Extensible records in a pure calcu-
lus of subtyping. Research report 81, DEC Sys-
tems Research Center, 1992. Also in [GM94].

typed object-oriented programming. In Proceed-
ings of Theoretical Aspects of Computer Software
(TACS’94), Sendai, Japan, volume 789 of LNCS,

[FM97] K. Fisher and J.C. Mitchell. On the relation-
ship between classes, objects, and data abstrac-
tion. In Proceedings of the International Sum-
er School on Mathematics of Program, LNCS.

[GM94] Carl A. Gunter and John C. Mitchell. Theo-
etical Aspects of Object-Oriented Programming:
Types, Semantics, and Language Design. The

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

[Has94] Tyu Hasegawa. Categorical data types in para-
metric polymorphism. Mathematical Structures in

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

[Jac96] Bart Jacobs. Objects and classes, co-
algebraically. In Object-Orientation with Par-
allelism and Persistence, pages 83–103. Kluwer

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

[Lis88] Barbara H. Liskov. Data abstraction and hierar-

A behavioral notion of subtyping. TOPLAS,

[LW95] Gary T. Leavens and William E. Weihl. Specifi-
cation and verification of object-oriented programs
using supertype abstraction. Acta Informatica,

[Mau95] Ian Maung. On simulation, subtyping and sub-
stitutability in sequential object systems. Formal


