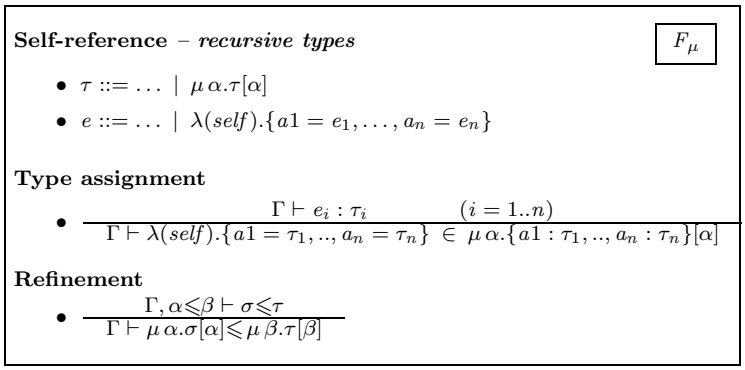
0.1 Self-reference

Recursive types are compound types in which the type itself occurs as the type of one of its components. Self-reference in objects clearly involves recursive types since the expression *self* denotes the object itself, and hence has the type of the object. In F_μ, our extension of $F_{\leqslant}$ taken from [CoHC90], recursive types are written as $\mu\,\alpha.\tau[\alpha]$, where μ is the recursion abstractor and α a type variable. The dependence of τ on α is made explicit by writing $\tau[\alpha]$. We will use the type expressions $\mu\,\alpha.\tau[\alpha]$ to type object specifications of the form $\lambda(self).\{a_1 = e_1, \ldots, a_n = e_n\}$ as indicated by the type assignment rule below. Object specifications may be regarded as class descriptions in C++ or Eiffel.



0-1

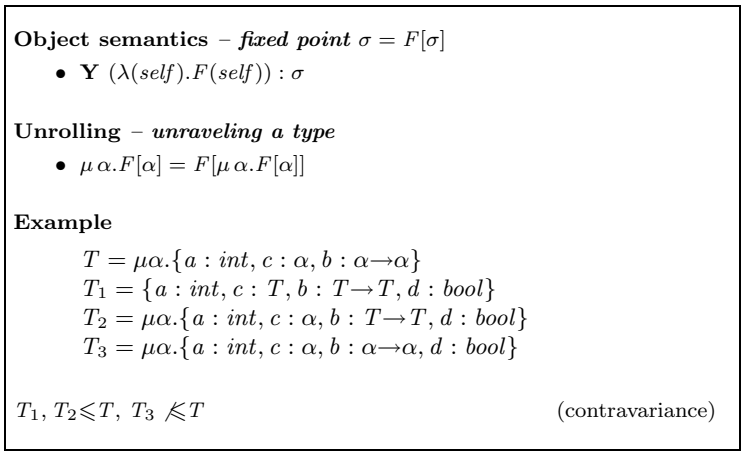
Slide 0-1: A calculus for recursive types

The subtype refinement rule for recursive types states that $\mu\,\alpha.\sigma[\alpha] \leqslant \mu\,\beta.\tau[\beta]$ if we can prove that $\sigma \leqslant \tau$ assuming that $\alpha \leqslant \beta$.

An object specification $\lambda(self).\{\ldots\}$ is a function with the type of the actual object as its domain and (naturally) also as its range. For convenience we will write an object specification as $\lambda(self).F$, where F denotes the object record, and the type of an object specification as $\mu\,\alpha.F[\alpha]$, where $F[\alpha]$ denotes the (abstract) type of the record F.

To obtain from an object specification $\lambda(self).F$ the object that it specifies, we need to find some type σ that types the record specification F as being of type σ precisely when we assign the expression *self* in F the type σ. Technically, this means that the object of type σ is a fixed point of the object specification $\lambda(self).F(self)$ which is of type $\sigma \to \sigma$. We write this as $\mathbf{Y}\ (\lambda(self).F(self)) : \sigma$, which says that the object corresponding to the object specification is of type σ. See slide **??**.

Finding the fixed point of a specification involves technically a procedure known as *unrolling*, which allows us to rewrite the type $\mu\,\alpha.F[\alpha]$ as $F[\mu\,\alpha.F[\alpha]]$. Notice that unrolling is valid, precisely because of the fixed point property. Namely, the object type σ is equal to $\mu\,\alpha.F[\alpha]$, due to the type assignment rule, and we have that $\sigma = F[\sigma]$. See slide **??**.



Slide 0-2: Recursive types – examples

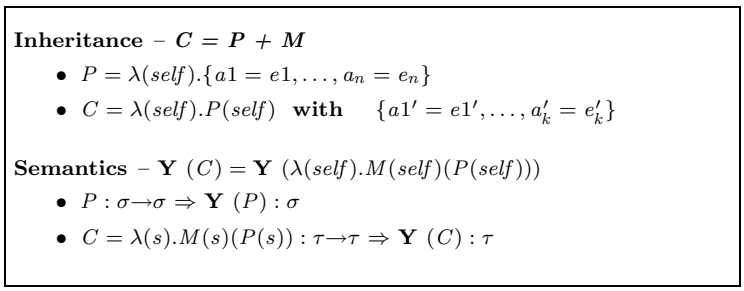
Unrolling allows us to reason on the level of types and to determine the subtyping relation between recursive subtypes. Consider, for example, the type declarations T and T_i $(i = 1..3)$ above. Based on the refinement rules for object records, functions and recursive types, we may establish that $T_1 \leqslant T$, $T_2 \leqslant T$ but $T_3 \not\leqslant T$. To see that $T_1 \leqslant T$, it suffices to substitute T for α in F, where $F = \{a : Int, c : \alpha, b : \alpha\rightarrow\alpha\}$. Since $F[T] = \{a : Int, c : T, b : T\rightarrow T\}$ we immediately see that T_1 only extends T with the field $d : Bool$, hence $T_1 \leqslant T$. A similar line of reasoning is involved to determine that $T_2 \leqslant T$, only we need to unroll T_2 as well. We must then establish that $c : T_2 \leqslant c : T$, which follows from an application of the refinement rule.

To show that $T_3 \not\leqslant T$, let $G[\beta] = \{a : Int, c : \beta, b : \beta\rightarrow\beta, d : Bool\}$ and $T_3 = \mu\,\beta.G[\beta]$. Then, by unrolling, $T_3 = G[T_3] = \{a : Int, c : T_3, b : T_3\rightarrow T_3, d : Bool\}$. Now, suppose that $T_3 \leqslant T$, then $G[T_3] \leqslant F[T_3]$ and consequently $b : T_3\rightarrow T_3$ must refine $b : T\rightarrow T$. But from the latter requirement it follows that $T_3 \leqslant T$ and that $T \leqslant T_3$ (by the contravariance rule for function subtyping). However, this leads to a contradiction since T is clearly not equal to T_3 because T_3 contains a field $d : Bool$ that does not occur in T.

Although analyses of this kind are to some extent satisfactory in themselves, the reader may wonder where this all leads to. In the following we will apply these techniques to show the necessity of dynamic binding and to illustrate that inheritance may easily violate the subtyping requirements.

Inheritance In section **??** we have characterized inheritance as an incremental modification mechanism, which involves a dynamic interpretation of the expression *self*. In the recursive type calculus F_μ we may characterize this more precisely, by regarding a derived object specification C as the result of applying the modifier M to the object specification P. We employ the notation $C = \lambda(self).P(self)$ **with** $\{a'_1 = e'_1, \ldots, a'_k = e'_k\}$ to characterize derivation by inheri-

tance, and we assume the modifier M corresponding with $\{a'_1 = e'_1, \ldots, a'_k = e'_k\}$ to extend the record associated with P in the usual sense. See slide **??**.

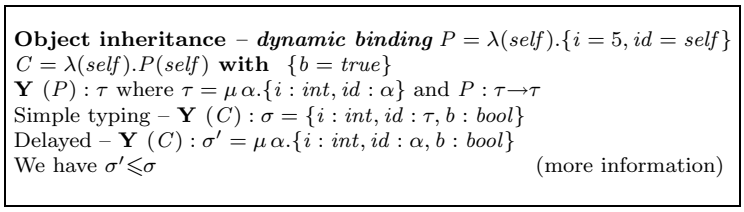


0-3

Slide 0-3: Inheritance semantics – self-reference

The meaning of an object specification C is again a fixed point $\mathbf{Y}\ (C)$, that is $\mathbf{Y}\ (\lambda(self).M(self)(P(self)))$. Now when we assume that the object specification is of type $\tau{\rightarrow}\tau$ (and hence $\mathbf{Y}\ (P)$ of type τ), and that C is of type $\sigma{\rightarrow}\sigma$ (and hence $\mathbf{Y}\ (C)$ of type σ), then we must require that $\sigma{\leqslant}\tau$ to obtain a properly typed derivation. We write $C{\leqslant}P$ whenever $\sigma{\leqslant}\tau$.

A first question that arises when we characterize inheritance as incremental modification is how we obtain the meaning of the composition of two object specifications.



0-4

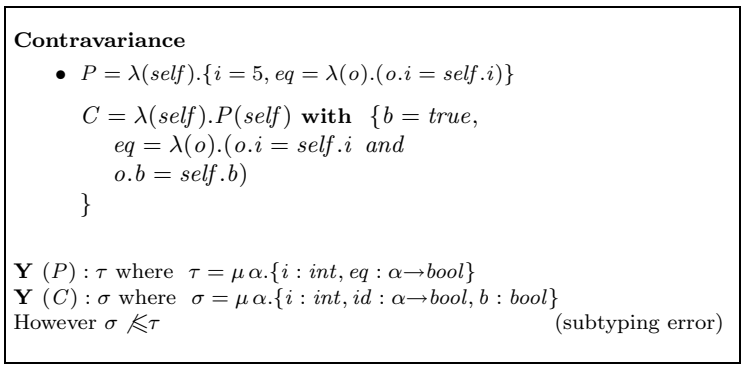
Slide 0-4: Object inheritance – dynamic binding

Let (parent) P and (child) C be defined as above. Now, if we know that the type of $\mathbf{Y}\ (P)$ is τ then we may simply characterize $\mathbf{Y}\ (C)$ as being of type $\sigma = \{i : Bool, id : \tau, b : Bool\}$. However, when we delay the typing of the P component (by first composing the record specifications before abstracting from *self*) then we may obtain $\sigma' = \mu\,\alpha.\{i : Int, id : \alpha, b : Bool\}$ as the type of $\mathbf{Y}\ (C)$. By employing the refinement rule and unrolling we can show that $\sigma'{\leqslant}\sigma$. Hence, delayed typing clearly provides more information and must be considered as the best choice. Note, however, that both $\sigma'{\leqslant}\tau$ and $\sigma{\leqslant}\tau$ hold. See slide **??**.

A second, important question that emerges with respect to inheritance is how self-reference affects the subtyping relation between object specifications related by inheritance.

Consider the object specifications P and C given in slide **??**. In the (derived) specification C, the method *eq* is redefined to include an equality test for the b

component. However, when we determine the object types corresponding to the specifications P and C we observe that $C \not\leqslant P$.



0-5

Slide 0-5: Object inheritance – contravariance

The reasoning is as follows. For $\mathbf{Y}\ (P) : \tau$ and $\mathbf{Y}\ (C) : \sigma$, we have that $\sigma = \mu\,\beta.\{i : Int, id : \beta \rightarrow Bool, b : Bool\}$ which is (by unrolling) equal to $\{i : Int, id : \sigma \rightarrow Bool, b : Bool\}$. Now suppose that $\sigma \leqslant \tau$, then we have that $\{i : Int, eq : \sigma \rightarrow Bool, b : Bool\}$ is a subtype of $\{i : Int, eq : \tau \rightarrow Bool\}$ which is true when $eq : \sigma \rightarrow Bool \leqslant eq : \tau \rightarrow Bool$ and hence (by contravariance) when $\sigma \leqslant \tau$. Clearly, this is impossible. Hence $\sigma \not\leqslant \tau$.

We have a problem here, since the fact that $C \not\leqslant P$ means that the type checker will not be able to accept the derivation of C from P, although C is clearly dependent on P. The solution to our problem lies in making the type dependency involved in deriving C from P explicit. Notice, in this respect, that in the example above we have omitted the type of the abstraction variable in the definition of *eq*, which would have to be written as $\lambda\, x : \mathbf{Y}\ (P).x.i = self.i$ (and in a similar way for C) to do it properly.

Type dependency The expression *self* is essentially of a polymorphic nature. To make the dependency of object specification on *self* explicit, we will employ an explicit type variable similar as in $F_{\leqslant}$.

Let $F[\alpha]$ stand for $\{a_1 : \tau_1, \ldots, a_n : \tau\}$ as before. We may regard $F[\alpha]$ as a type function, in the sense that for some type τ the expression $F[\tau]$ results in a type. To determine the type of an object specification we must find a type σ that satisfies both $\sigma \leqslant F[\sigma]$ and $F[\sigma] \leqslant \sigma$.

We may write an object specification as $\Lambda\alpha \leqslant F[\alpha].\, \lambda(self : \alpha).\{a_1 = e_1, \ldots, a_n = e_n\}$, which is typed as $\forall\, \alpha \leqslant F[\alpha].\alpha \rightarrow F[\alpha]$. The constraint that $\alpha \leqslant F[\alpha]$, which is called an *F-bounded constraint*, requires that the subtype substituted for α is a (structural) refinement of the record type $F[\alpha]$. As before, we have that $\mathbf{Y}\ (P[\sigma]) = \sigma$ with $\sigma = \mu\,\alpha.F[\alpha]$, which differs from our previous definition only by making the type dependency in P explicit. See slide **??**.

Now, when applying this extended notion of object specification to the char-

0-6

Type dependency – ***is polymorphic***

- Let $F[\alpha] = \{m_1 : \sigma_1, \ldots, m_j : \sigma_j\}$
- $P : \forall\, \alpha \leqslant F[\alpha].t \rightarrow F[\alpha]$
- $P = \Lambda\alpha \leqslant F[\alpha].\, \lambda(self : \alpha).\{m_1 : e1, \ldots, m_j : e_j\}$

F-bounded constraint $\alpha \leqslant F[\alpha]$
Object instantiation: $\mathbf{Y}\ (P[\sigma])$ for $\sigma = \mu\, t.F[t]$
We have $P[\sigma] : \sigma \rightarrow F[\sigma]$ because $F[\sigma] = \sigma$

Slide 0-6: Bounded type constraints

acterization of inheritance, we may relax our requirement that $\mathbf{Y}\ (C)$ must be a subtype of $\mathbf{Y}\ (P)$ into the requirement that $G[\alpha] \leqslant F[\alpha]$ for any α, where F is the record specification of P and G the record specification of C.

0-7

Inheritance

$$P = \Lambda\alpha \leqslant F[\alpha].\lambda(self : \alpha).\{\ldots\}$$
$$C = \Lambda\alpha \leqslant G[\alpha].\lambda(self : \alpha).P[\alpha](self)\ \textbf{with}\ \ \{\ldots\}$$

with recursive types
$F[\alpha] = \{i : int, id : \alpha \rightarrow bool\}$
$G[\alpha] = \{i : int, id : \alpha \rightarrow bool, b : bool\}$
Valid, because $G[\alpha] \leqslant F[\alpha]$
However $\mathbf{Y}\ (C[\sigma]) \not\leqslant_{subtype} \mathbf{Y}\ (P[\tau])$

Slide 0-7: Inheritance and constraints

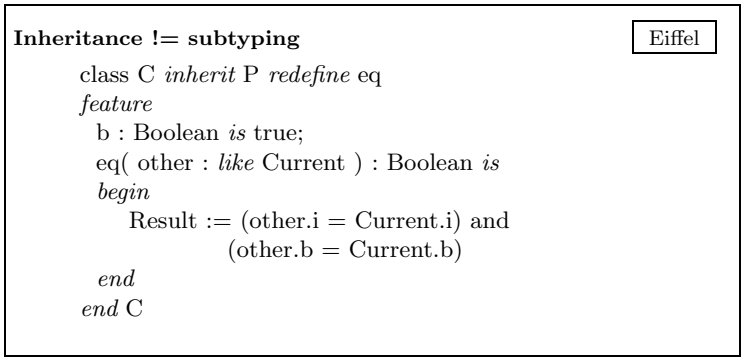
For example, when we declare $F[\alpha]$ and $G[\alpha]$ as in slide **??**, we have that $G[\alpha] \leqslant F[\alpha]$ for every value for α. However, when we find types σ and τ such that $\mathbf{Y}\ (C[\sigma]) : \sigma$ and $\mathbf{Y}\ (P[\tau]) : \tau$ we (still) have that $\sigma \not\leqslant \tau$. Conclusion, inheritance allows more than subtyping. In other words, our type checker may guard the structural application of inheritance, yet will not guarantee that the resulting object types behaviorally satisfy the subtype relation.

Discussion – Eiffel is not type consistent We have limited our exploration of the recursive structure of objects to (polymorphic) object variables. Self-reference, however, may also occur to *class variables*. The interested reader is referred to [CoHC90]. The question that interests us more at this particular point is what benefits we may have from the techniques employed here and what lessons we may draw from applying them.

One lesson, which should not come as a surprise, is that a language may allow us to write programs that are accepted by the compiler yet are behaviorally incorrect. However, if we can determine syntactically that the subtyping relations between classes is violated we may at least expect a warning from the compiler.

So one benefit, possibly, is that we may improve our compilers on the basis of the type theory presented in this chapter. Another potential benefit is that we may better understand the trade-offs between the particular forms of polymorphism offered by our language of choice.

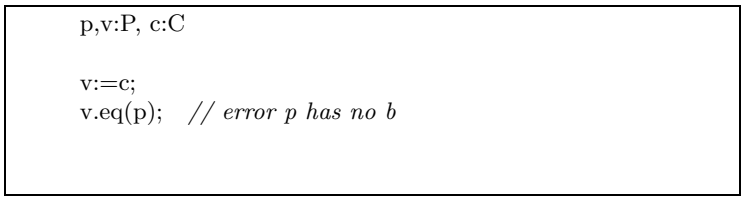
The analysis given in [CoHC90] indeed leads to a rather surprising result. Contrary to the claims made by its developer, [CoHC90] demonstrate that Eiffel is *not* type consistent. The argument runs as follows. Suppose we define a class C with a method *eq* that takes an argument of a type similar to the type of the object itself (which may be written in Eiffel as *like Current*). We further assume that the class P is defined in a similar way, but with an integer field i and a method *eq* that tests only on i. See slide **??**.



0-8

Slide 0-8: Inheritance and subtyping in Eiffel

We may then declare variables v and p of type P. Now suppose that we have an object c of type C, then we may assign c to v and invoke the method *eq* for v, asking whether p is equal to v, as in

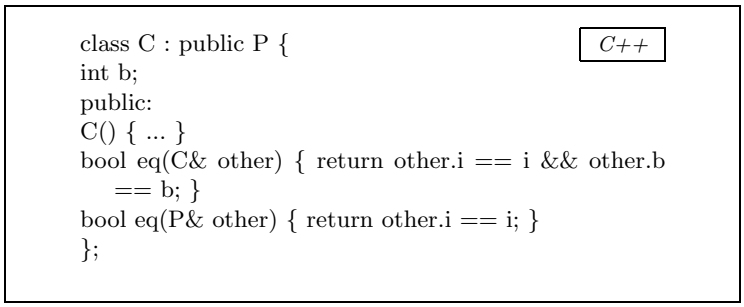


0-9

Slide 0-9: Example

Since v is associated with an instance of C, but syntactically declared as being of type P, the compiler accepts the call. Nevertheless, when p is associated with an instance of P trouble will arise, since (due to dynamic binding) the method *eq* defined for C will be invoked while p not necessarily has a field b.

When we compare the definition of C in Eiffel with how we may define C in C++, then we are immediately confronted with the restriction that we do not have such a dynamic typing mechanism as *like Current* in C++. Instead, we may use overloading, as shown in slide **??**.



Slide 0-10: Inheritance and subtyping in C++

When we would have omitted the *P* variant of *eq*, the compiler complains about hiding a virtual function. However, the same problem arises when we define *eq* to be virtual in *P*, unless we take care to explicitly cast *p* into either a *C* or *P* reference. (Overloading is also used in [Liskov93] to solve a similar problem.) In the case we choose for a non-virtual definition of *eq*, it is determined statically which variant is chosen and (obviously) no problem occurs.

Considering that determining equality between two objects is somehow orthogonal to the functionality of the object proper, we may perhaps better employ externally defined overloaded functions to express relations between objects. This observation could be an argument to have overloaded functions apart from objects, not as a means to support a hybrid approach but as a means to characterize relations between objects in a type consistent (polymorphic) fashion.