On Subtyping and Matching

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract. A relation between recursive object types, called *matching*, has been proposed as a generalization of subtyping. Unlike subtyping, matching does not support subsumption, but it does support inheritance of binary methods. We argue that matching is a good idea, but that it should not be regarded as a form of F-bounded subtyping (as was originally intended). We show that a new interpretation of matching as higher-order subtyping has better properties. Matching turns out to be a third-order construction, possibly the only one to have been proposed for general use in programming.

1 Introduction

Subtyping is one of the most basic and best understood concepts in object-oriented programming. The subtyping relation can be defined rather naturally for first-order constructions such as record types, object types, function types, and recursive types [9, 16]. It can be extended to second-order constructions such as quantifiers for polymorphism and for data abstraction [19, 23], and to type operators [21, 35]. In programming, subtyping is the basis for both subsumption and inheritance.

Unfortunately, the subtyping relation does not hold between some recursively defined object types that arise commonly [15, 24]. This failure of subtyping is necessary for soundness. The standard, pragmatic solution is to weaken type definitions so that subtyping is achieved, and to use dynamic type tests to recover lost type information (as for example in Modula-3 [34]). A better solution is to employ Self types (as in [4, 5]), but this helps only for covariant occurrences of recursion variables (e.g., not for binary methods [11]).

Recently, a relation between recursive object types, called *matching*, has been proposed to circumvent this problem [14]. Unlike subtyping, matching does not support subsumption, but it does support inheritance of binary methods and parameterization. The original interpretation of matching was in terms of F-bounded subtyping. We show that a new interpretation of matching as higher-order subtyping has better properties, and produces the expected typing rules.

In the next section we explain the basic issues by means of examples and we hint at a solution based on the informal concept of protocol. In section 3 we introduce the matching relation, which is intended to capture the notion of protocol extension. In section 4 we present type operators, which are a formal counterpart for protocols. In the following two sections, we use type operators to explain matching. In section 5 we discuss a tentative formalization of matching in terms of F-bounded subtyping. In section 6 we present a more satisfactory formalization of matching in terms of higher-order subtyping. In section 7 we apply our techniques to express classes and inheritance.

2 The Power and Limits of Subtyping

In this section we introduce two examples that we use throughout the paper. The first example is that of two types in the subtyping relation. Thanks to subtyping, simple techniques allow us to express classes with inheritance. In the second example, subtyping fails. We describe informally the notion of protocol extension, which is a possible replacement for subtyping.

2.1 Objects Types and Subtyping

We introduce our notation and some basic typing rules in the context of our first example. This example illustrates how subtyping works.

We consider two types Inc and IncDec containing an integer field and some methods:

$$\begin{aligned} \mathrm{Inc} &\triangleq \mu(X)[n{:}\mathrm{Int},\ inc^{+}{:}X] \\ \mathrm{IncDec} &\triangleq \mu(Y)[n{:}\mathrm{Int},\ inc^{+}{:}Y,\ dec^{+}{:}Y] \end{aligned}$$

A type of the form $\mu(X)A$ is a recursive type. For now we assume that a recursive type and its unfolding are equivalent, in the sense that we consider $\mu(X)A\{X\} = A\{\mu(X)A\{X\}\}$. We refer to this as the *unfolding property* of recursive types. Thus, we have $\mathrm{Inc} = [n{:}\mathrm{Int}, inc^{+}{:}\mathrm{Inc}]$ and $\mathrm{IncDec} = [n{:}\mathrm{Int}, inc^{+}{:}\mathrm{IncDec}, dec^{+}{:}\mathrm{IncDec}]$.

A type of the form $[v_i{:}B_i\ ^{i \in I}, m_j{}^{+}{:}C_j\ ^{j \in J}]$ is an object type, where the v_i are updatable fields and the m_j are methods; we always indicate a method type by a $^{+}$ sign. For example, the type $[n{:}\mathrm{Int}, inc^{+}{:}\mathrm{Inc}]$ has a field n of type Int and a method inc with result type Inc.

A typical object of type Inc is:

$$\begin{aligned} &p : \mathrm{Inc} \triangleq \\ &\quad [n = 0, \\ &\quad\ inc = \varsigma(self{:}\ \mathrm{Inc})\ self.n := self.n + 1] \end{aligned}$$

Here p : Inc means that p has type Inc. The binder ς binds the self variable of the method inc. The code self.n := self.n +1 is the body of the method inc; the method returns the updated self. Update may be either functional (returning a new object) or imperative (modifying self in place); either semantics will do for the purposes of this paper.

The method inc can be understood roughly as the function λ(self: Inc) self.n := self.n +1, of type Inc→Inc. However, we prefer to avoid the λ notation since the method is never explicitly applied to an argument. Instead, the method is invoked by writing p.inc; when p.inc is executed, p is bound to the self parameter.

Subtyping (<:) is a reflexive and transitive relation on types. For recursive types we have the rule:

$$\mu(X)A\{X\} <: \mu(Y)B\{Y\}$$
$$\text{if } X <: Y \text{ implies } A\{X\} <: B\{Y\}$$

Two recursive types are in the subtyping relation when their bodies are in the subtyping relation under the assumption that the recursion variables are in the subtyping relation [9].

For object types, we have the rule:

$$[v_i{:}B_i\ ^{i\in I}, m_j^+{:}C_j\ ^{j\in J}] <: [v_i{:}B_i\ ^{i\in I'}, m_j^+{:}C_j'\ ^{j\in J'}]$$
$$\text{if } C_j <: C_j' \text{ for all } j\in J', \text{ with } I'\subseteq I \text{ and } J'\subseteq J$$

A longer object type is a subtype of a shorter one. In addition, the result type of a method may be smaller in a subtype than in a supertype; the type of a field must be the same [2, 5]. The latter condition (the invariance of field types) is necessary for soundness, because fields can be updated.

Object types are often defined recursively; therefore, it is convenient to regard the combination of a recursive type and an object type as a single construction of the form:

$$\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j^+{:}C_j\{X\}\ ^{j\in J}]$$

In this view, the recursion variable X is often called "MyType" or "Self": it is the type of the self parameter of methods. Combining the previous rules for subtyping, we obtain the following derived rule for recursive object types:

$$\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j^+{:}C_j\{X\}\ ^{j\in J}] <: \mu(Y)[v_i{:}B_i\ ^{i\in I'}, m_j^+{:}C_j'\{Y\}\ ^{j\in J'}]$$
$$\text{if } X <: Y \text{ implies } C_j\{X\} <: C_j'\{Y\} \text{ for all } j\in J', \text{ with } I'\subseteq I \text{ and } J'\subseteq J$$

By applying this derived rule to our example, we obtain:

$$\text{IncDec} <: \text{Inc}$$

The subtyping relation between IncDec and Inc allows *subsumption*: we can treat a member of IncDec as a member of Inc. The general rule for subsumption is:

$$\text{if } \quad a : A \quad \text{and} \quad A <: B \quad \text{then} \quad a : B$$

2.2 Inheritance

The subtyping relation between IncDec and Inc plays an important role in inheritance. From our perspective, inheritance is obtained by writing sufficiently polymorphic code fragments that can be instantiated and reused for implementing objects of several types. We can express the polymorphism of these fragments with bounded universal quantification. For example, consider a function that increments the field n of its argument and returns the modified argument; this function works for arguments of type Inc, and also for arguments of subtypes of Inc. This is written:

$$\begin{aligned}&\text{pre-inc} : \forall(X{<:}\text{Inc})X\rightarrow X \ \triangleq\\ &\quad \lambda(X{<:}\text{Inc})\ \lambda(\text{self}{:}X)\ \text{self.n} := \text{self.n}+1\end{aligned}$$

To typecheck the body of pre-inc, we use rules implying that if X <: Inc and self:X then self.n:Int and, for b:Int, self.n:=b : X. (See [5] for a discussion of these rules.)

We call a code fragment such as pre-inc a *pre-method*. By polymorphic specialization of pre-inc we can obtain code suitable for implementing the method inc in an object of type Inc or IncDec:

$$\begin{aligned}&\text{pre-inc(Inc)} : \text{Inc}\rightarrow\text{Inc}\\ &\text{pre-inc(IncDec)} : \text{IncDec}\rightarrow\text{IncDec}\end{aligned}$$

Thus, the generic pre-method pre-inc, which was presumably first written for building objects of type Inc, can be reused for building objects of type IncDec, because IncDec <: Inc. The reuse does not require retypechecking the pre-method.

2.3 Classes

Pre-method reuse can be systematized by assembling collections of pre-methods into *classes*. A class for an object type A can be described as a collection of pre-methods and initial field values, plus a way of generating new objects of type A. Such a collection could be taken as primitive; instead, we choose to represent classes as objects [2, 5]. Although it may be mildly confusing at first, this reduction of classes to objects has the advantage of making explicit the typing properties of methods and of object constructors.

In a class for an object type A, the pre-methods are parameterized over all subtypes of A, so that they can be reused (inherited) by any class for any subtype of A. Let A be a type of the form $\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j^{+}{:}C_j\{X\}\ ^{j\in J}]$. As part of a class for A, a pre-method for m_j would be stored in a field of type $\forall(X{<:}A)X\rightarrow C_j\{X\}$.

For example, IncClass and IncDecClass are the types of classes for Inc and IncDec, respectively:

```
IncClass  ≜
      [new⁺: Inc,
       n: Int,
       inc: ∀(X<:Inc)X→X]
IncDecClass  ≜
      [new⁺: IncDec,
       n: Int,
       inc: ∀(X<:IncDec)X→X,
       dec: ∀(X<:IncDec)X→X]
```

A typical class of type IncClass reads:

```
incClass : IncClass  ≜
      [new = ς(classSelf: IncClass)
            [n = classSelf.n,
             inc = ς(self:Inc) classSelf.inc(Inc)(self)]
       n = 0,
       inc = pre-inc]
```

The code for new is regular: it assembles all the pre-methods and initial field values of the class into a new object. The variable classSelf denotes the incClass object, and classSelf.inc is pre-inc.

Inheritance is obtained by extracting a pre-method from a class and reusing it for constructing another class. For example, the pre-method pre-inc of type ∀(X<:Inc)X→X in a class for Inc could be extracted and reused as a pre-method of type ∀(X<:IncDec)X→X in a class for IncDec:

```
incDecClass : IncDecClass  ≜
      [new = ς(classSelf: IncDecClass)[...],
       n = 0,
       inc = incClass.inc,
       dec = ...]
```

This example of inheritance requires the subtyping ∀(X<:Inc)X→X <: ∀(X<:IncDec)X→X, which follows from the subtyping rules for quantified types and function types:

```
∀(X<:A)B <: ∀(X<:A′)B′       if A′<:A and if X<:A implies B<:B′
A→B <: A′→B′                 if A′ <: A and B <: B′
```

In summary, inheritance from a class for Inc to a class for IncDec is enabled by the subtyping IncDec <: Inc. Unfortunately, inheritance is possible and desirable even in situations where such subtypings do not exist. As we shall see in the rest of the paper, a more general treatment of inheritance requires a more sophisticated treatment of classes, based on relations between object types other than subtyping.

2.4 Binary Methods

We now consider an example similar to the one of Inc and IncDec, but with binary methods. The first parameter of a binary method is self and the second parameter must have the same type as self. In our example, we consider a recursive object type Max, with a field n and a binary method max. Our intent is that max takes two objects (self and other) and returns the object that has the greatest value for the field n.

$$\text{Max} \triangleq \mu(X)[\text{n:Int}, \text{max}^{+}\text{:}X\rightarrow X]$$

We consider also a type MinMax with an additional binary method min:

$$\text{MinMax} \triangleq \mu(Y)[\text{n:Int}, \text{max}^{+}\text{:}Y\rightarrow Y, \text{min}^{+}\text{:}Y\rightarrow Y]$$

When we attempt to reproduce the development of the previous sections, we immediately run into a difficulty: the subtyping MinMax <: Max does not hold according to the rules we have adopted. Moreover, it would be unsound to ignore this failure, as it is easy to construct examples where allowing MinMax <: Max leads to run-time errors.

Technically, an attempt to prove MinMax <: Max fails at the point where we try to verify that $X\rightarrow X <: Y\rightarrow Y$ under the assumption that $X <: Y$. Since the occurrences of X and Y on the left of $\rightarrow$ are contravariant, we would need $Y <: X$, which is the opposite of the assumption we have available.

We may now observe that the development in section 2.1 relies on the fact that the type for the method inc is covariant with respect to the recursion variable that represents Self. Whenever this covariance condition is violated, as with binary methods, a longer recursive object type is not a subtype of a shorter recursive object type.

The failure of subtyping has dire consequences beyond the loss of subsumption. We can still parameterize over the subtypes of Max, for example writing a pre-method for max:

```
pre-max : $\forall(X<:\text{Max})X\rightarrow X\rightarrow X \triangleq$
    $\lambda(X<:\text{Max})\ \lambda(\text{self}:X)\ \lambda(\text{other}:X)$
        if self.n>other.n then self else other
```

However, this parameterization is of no benefit, since the pre-method cannot be specialized at type MinMax. Similarly, we lose inheritability of pre-methods: since $\forall(X<:\text{Max})X\rightarrow X$ is not a subtype of $\forall(X<:\text{MinMax})X\rightarrow X$, the pre-method pre-max cannot be reused in building a class for MinMax.

The lack of inheritance is most disappointing, because the code for pre-max can in fact be given any type of the form:

$$\forall(X<:\mu(Y)[n:Int,\ max^{+}:Y\rightarrow Y,\ ...\ \textit{(anything)}\ ...\ ])X\rightarrow X\rightarrow X$$

Without inheritance, we may have to rewrite the code for pre-max for many types of this form, retypechecking it in each case. In particular, we may have to write:

```
pre-max′ : ∀(X<:MinMax)X→X→X  ≜
    λ(X<:MinMax) λ(self:X) λ(other:X)
        if self.n>other.n then self else other
```

The body of the pre-method (namely, if self.n>other.n then self else other) does not change. We would be able to reuse pre-max if only we could write down an appropriate polymorphic type for it.

2.5 Protocols

Our difficulties with inheritance can be traced back to the fact that MinMax is not a subtype of Max. Although there is no hope of forcing this subtyping, there is hope of finding some other useful formal relationship between these types. In this section we introduce the notion of *protocol,* in order to capture an intuitive relationship between Max and MinMax. Our discussion is informal, but we will soon see how to formalize the notions described here.

The protocol of an object characterizes its interface, much like the type of the object; the protocol consists of the names of fields and methods of the object, together with their types. The protocol can be derived from the type of the object, and differs from it only by the absence of recursion. Informally, the protocol of a recursive object type $\mu(X)[v_i:B_i{}^{i\in I},\ m_j{}^{+}:C_j\{X\}{}^{j\in J}]$ is the collection of names and types:

$$v_i:B_i{}^{i\in I},\ m_j{}^{+}:C_j\{\mathbf{Self}\}{}^{j\in J}$$

For now, we use **Self** as a keyword, without pondering what it means. The protocols of Max and MinMax are:

$$\text{MaxProtocol} \quad \triangleq \quad n:Int,\ max^{+}:\mathbf{Self}\rightarrow\mathbf{Self}$$
$$\text{MinMaxProtocol} \quad \triangleq \quad n:Int,\ max^{+}:\mathbf{Self}\rightarrow\mathbf{Self},\ min^{+}:\mathbf{Self}\rightarrow\mathbf{Self}$$

We introduce also a *protocol-extension* relation, with the following tentative definition:

$$v_i:B_i{}^{i\in I},\ m_j{}^{+}:C_j\{\mathbf{Self}\}{}^{j\in J} \quad \textit{extends} \quad v_i:B_i{}^{i\in I'},\ m_j{}^{+}:C_j'\{\mathbf{Self}\}{}^{j\in J'}$$
$$\text{if } C_j\{\mathbf{Self}\} <: C_j'\{\mathbf{Self}\} \text{ for all } j\in J', \text{ with } I'\subseteq I \text{ and } J'\subseteq J$$

In this definition, we treat **Self** as a type (and assume, for example, that **Self** <: **Self**). We may now assert that the protocol of MinMax extends the protocol of Max:

$$\text{MinMaxProtocol} \quad \textit{extends} \quad \text{MaxProtocol}$$

This is not the same as asserting (incorrectly) that MinMax is a subtype of Max.

When properly formalized, the notion of protocol extension could be used in parameterization, in place of subtyping. For example, when parameterizing over a type X, we would like to be able to express the constraint that the protocol of X extends MaxProtocol, meaning that the protocol of X includes a field n of type Int and a method max of type **Self**→**Self**. We could then say that, for every such X, the function pre-max has type X→X→X. We will see how to express this constraint as an appropriate polymorphic type, and will be able to write pre-max so that it is inheritable.

3 Matching

There have been several theoretical and practical approaches that capture concepts similar to that of protocol [15, 18, 22, 25, 26, 28, 29, 30, 31, 33]. Recently, Bruce *et al.* [10, 14] proposed axiomatizing the protocol-extension relation as if it were a relation between recursive object types, called *matching*. In this section we start considering that proposal. We identify some basic properties of matching and then show how we use matching for inheritance of binary methods.

3.1 Basic Properties of Matching

We write A <# B to mean that A matches B; that is, that the protocol of A extends the protocol of B. We do not yet consider how to prove the assertion A <# B, but we expect to have, for example:

IncDec <# Inc
MinMax <# Max

In particular, we may write X <# A, where X is a variable. We may then quantify over all types that match a given one, as follows:

∀(X<#A)B{X}

We call ∀(X<#A)B *match-bounded quantification*, and say that occurrences of X in B are *match-bound*.

Using match-bounded quantification, we can rewrite the polymorphic function pre-inc in terms of matching rather than subtyping:

pre-inc : ∀(X<#Inc)X→X ≜
    λ(X<#Inc) λ(self:X) self.n := self.n+1

pre-inc(IncDec) : IncDec→IncDec

It remains to see why this code fragment typechecks. Some of the necessary properties are:

```
if X<#Inc and x:X then x.n : Int
if X<#Inc and x:X and b:Int then x.n:=b : X
```

Similarly, we can write a polymorphic version of the function pre-max:

```
pre-max : ∀(X<#Max)X→X→X  ≜
    λ(X<#Max) λ(self:X) λ(other:X)
        if self.n>other.n then self else other

pre-max(MinMax) : MinMax→MinMax→MinMax
```

For typechecking this code fragment, we need:

```
if X<#Max and x:X then x.n : Int
```

Thus, the use of match-bounded quantification enables us to express the polymorphism of pre-max. The treatment of pre-max is not substantially different from that of pre-inc: contravariant and covariant occurrences of Self are treated uniformly.

We have not used subsumption in either of these examples. In fact, since we think of matching as protocol extension, we do not expect a subsumption-like property to hold for matching:

a : A   and   A <# B   need not imply   a : B

The properties that we have attributed to matching must hold regardless.

The absence of a subsumption-like property can pose a problem. Knowing that A <# B is not quite as good as knowing that A <: B. For example, imagine that we forget that IncDec <: Inc and attempt to get by with IncDec <# Inc. Then we could not typecheck code such as:

```
inc : Inc→Inc  ≜
    λ(x:Inc) x.n := x.n+1

λ(x:IncDec) inc(x)
```

where we apply a function of type Inc→Inc to an object of type IncDec. We can circumvent this difficulty by turning inc into a polymorphic function of type ∀(X<#Inc)X→X:

```
pre-inc : ∀(X<#Inc)X→X  ≜
    λ(X<#Inc) λ(x:X) x.n := x.n+1

λ(x:IncDec) pre-inc(IncDec)(x)
```

Thus, at least in this example, we can do without subtyping (and subsumption), provided that we have the foresight of introducing sufficient parameterization. However, this parameterization may be cumbersome.

Despite this example, it is not clear whether matching can completely replace subtyping; we leave this as an open issue. We do not regard matching as a replacement for subtyping, but rather as a complement to subtyping.

3.2 Classes with Matching

We can now revise our treatment of classes from section 2.3, adapting it for matching. A class for an object type A can again be described as a collection of pre-methods and initial field values, plus a way of generating new objects of type A. The pre-methods are now parameterized over all object types matching A, so that they can be inherited by any class for an object type matching A. For example, MaxClass and MinMaxClass are the types for classes for Max and MinMax, respectively:

```
MaxClass  ≜
     [new⁺: Max,
      n: Int,
      max: ∀(X<#Max)X→X→X]
MinMaxClass  ≜
     [new⁺: MinMax,
      n: Int,
      max: ∀(X<#MinMax)X→X→X,
      min: ∀(X<#MinMax)X→X→X]
```

A typical class of type MaxClass reads:

```
maxClass : MaxClass  ≜
     [new = ς(classSelf: MaxClass)
            [n = classSelf.n,
             max = ς(self:Max) classSelf.max(Max)(self)],
      n = 0,
      max = pre-max]
```

The code for new typechecks assuming that Max <# Max, so that classSelf.max(Max)(self) : Max→Max. The pre-method pre-max is as given in section 3.1.

A typical class of type MinMaxClass reads:

```
minMaxClass : MinMaxClass  ≜
     [new = ς(classSelf: MinMaxClass)[...],
      n = 0,
      max = maxClass.max,
      min = ...]
```

The implementation of max is taken from maxClass, that is, it is inherited. The inheritance typechecks assuming that ∀(X<#Max)X→X→X <: ∀(X<#Min-

Max)X→X→X. Thus, we are still using some subtyping and subsumption as a basis for inheritance.

3.3 Discussion

In light of these examples, we may conclude that the idea of a matching relation is attractive:

- The fact that MinMax matches Max is reasonably intuitive, since the textual definition of MinMax does in fact match the textual definition of Max (with Self matching Self, and ignoring the additional components of MinMax). If anything, it is counterintuitive that subtyping should fail.
- Matching handles contravariant occurrences of Self and inheritance of binary methods.
- Matching is meant to be directly axiomatized as a relation between types, so the typing rules of a programming language that includes matching can be explained directly.
- Matching is simple from the programmer's point to of view, in comparison with more elaborate type-theoretic mechanisms that could be used in its place.

By examining examples, we have identified a few necessary properties of matching. However, we still have to settle on the exact typing rules for matching. These rules turn out to be different in proposed languages such as TOOPLE [10] and PolyTOIL [14]. The differences are subtle but fundamental. For example, in PolyTOIL matching is a transitive relation, while in TOOPLE it is not. We may further ask whether matching should be reflexive, or invariant under substitution. These questions, and more, inevitably arise in trying to design a programming language based on a matching relation.

We return to these questions in sections 5 and 6, where we consider two alternative formalizations of matching.

4 Type Operators

As we discussed in section 2, we are looking for formal relationships between the types Max and MinMax. These relationships might suggest a precise definition of matching, clarifying what we mean by the assertion MinMax <# Max; they might also suggest alternatives to matching.

In this section we introduce a theory of type operators that will enable us to express formal relationships between types.

4.1 Type Operators and Recursive Object Types

A type operator is a function from types to types [27]. We use the notations and rules of the lambda calculus for type operators: we write $\lambda(X)B\{X\}$ for the type operator that maps each type X to a corresponding type $B\{X\}$, write $F(A)$ for the application of the type operator F to the type A, and let $(\lambda(X)B\{X\})(A) = B\{A\}$.

We often move back and forth between type operators and their fixpoints. In one direction, we write F^* for the fixpoint of the type operator F. In the opposite direction, we have the notation A_{Op}:

$$\begin{array}{lll} F^* & \text{abbreviates} & \mu(X)F(X) \\ A_{Op} & \text{abbreviates} & \lambda(X)D\{X\} \quad \text{whenever } A \equiv \mu(X)D\{X\} \end{array}$$

When $D\{X\}$ is an object type $[v_i:B_i{}^{i\in I}, m_j{}^+:C_j\{X\}{}^{j\in J}]$, the type operator A_{Op} is a formalization of the protocol of A. In this case A_{Op} is $\lambda(X)[v_i:B_i{}^{i\in I}, m_j{}^+:C_j\{X\}{}^{j\in J}]$; this type operator is analogous to the protocol $v_i:B_i{}^{i\in I}, m_j{}^+:C_j\{\mathbf{Self}\}{}^{j\in J}$, but differs in being fully formal (e.g., it does not mention the keyword **Self**). In the example of Max and MinMax, we obtain:

$$\begin{array}{lll} Max_{Op} & \equiv & \lambda(X)[n{:}Int, max^+{:}X\to X] \\ MinMax_{Op} & \equiv & \lambda(Y)[n{:}Int, max^+{:}Y\to Y, min^+{:}Y\to Y] \end{array}$$

The unfolding property of recursive types yields:

$$\begin{array}{lllllll} Max_{Op}{}^* & = & \mu(X)\, Max_{Op}(X) & = & \mu(X)\,[n{:}Int, max^+{:}X\to X] & = & Max \\ Max_{Op}{}^* & = & Max_{Op}(\mu(X)\, Max_{Op}(X)) & = & Max_{Op}(Max) & & \end{array}$$

Note that A_{Op} is defined in terms of the syntactic form $\mu(X)D\{X\}$ of A. In particular, the unfolding $D\{A\}$ of A is not necessarily in a form such that $D\{A\}_{Op}$ is defined. Even if $D\{A\}_{Op}$ is defined, it need not equal A_{Op}. For example, consider:

$$\begin{array}{lllll} D\{X\} & \triangleq & \mu(Y)\, X\to Y & & \\ A & \triangleq & \mu(X)\, D\{X\} & & \\ D\{A\} & \equiv & \mu(Y)\, A\to Y & = A & \text{(by the unfolding property)} \\ A_{Op} & \equiv & \lambda(X)\, D\{X\} & & \\ D\{A\}_{Op} & \equiv & \lambda(Y)\, A\to Y & \neq A_{Op} & \end{array}$$

Thus, we may have two types A and B such that $A = B$ but $A_{Op} \neq B_{Op}$. This is a sign of trouble to come.

4.2 F-bounded Subtyping

F-bounded subtyping [15] was invented to support parameterization in the cases where simple subtyping is not sufficient. In the F-bounded approach, the property $A <: B_{Op}(A)$ is seen as a statement that the protocol of A extends the protocol of B.

We can explain this approach by reference to our examples. One common property of the types Max and MinMax is that they are both post-fixpoints of

Max_{Op}, that is:

$$\begin{array}{ll} Max <: Max_{Op}(Max) & (= Max) \\ MinMax <: Max_{Op}(MinMax) & (= [n:Int, max^{+}:MinMax\to MinMax]) \end{array}$$

As we said, Max_{Op} is a formalization of the protocol of Max. The property $X <: Max_{Op}(X)$ is seen as asserting that the protocol of X extends the protocol of Max. This view is justified because a recursive object type A such that $A <: [n:Int, max^{+}:A\to A]$ often has the shape $\mu(Y)[n:Int, max^{+}:Y\to Y, ...]$.

Since Max and MinMax are both post-fixpoints of Max_{Op}, it is useful to parameterize over all types X with the property that $X <: Max_{Op}(X)$. Thus, a function of type:

$$\forall(X<:Max_{Op}(X))B\{X\}$$

can be instantiated for the types Max and MinMax, with instances of types B{Max} and B{MinMax}. In particular, this form of parameterization leads to a general typing of pre-max, and permits the inheritance of pre-max:

$$\begin{array}{l} \text{pre-max} : \forall(X<:Max_{Op}(X))X\to X\to X \ \triangleq \\ \quad \lambda(X<:Max_{Op}(X))\ \lambda(self:X)\ \lambda(other:X) \\ \qquad \text{if self.n>other.n then self else other} \end{array}$$

$$\begin{array}{l} \text{pre-max}(Max) : Max\to Max\to Max \\ \text{pre-max}(MinMax) : MinMax\to MinMax\to MinMax \end{array}$$

The code of pre-max is well-typed because $X <: Max_{Op}(X)$ and self:X imply self : $[n:Int, max^{+}:X\to X]$ by subsumption; similarly, other : $[n:Int, max^{+}:X\to X]$. The instantiation pre-max(Max) is well-typed because $Max = Max_{Op}(Max)$.

As this example shows, the unfolding property of recursive types is critical. In general, given a function f : $\forall(X<:A\{X\})B\{X\}$, we would expect to be able to write $f(\mu(X)A\{X\})$; but $f(\mu(X)A\{X\})$ is well-typed only if $\mu(X)A\{X\} <: A\{\mu(X)A\{X\}\}$, and this condition holds naturally if $\mu(X)A\{X\} = A\{\mu(X)A\{X\}\}$.

Quantification of the form $\forall(X<:A\{X\})B\{X\}$ is called F-bounded quantification; it is characterized by the constraint $X <: A\{X\}$. This form of quantification is peculiar in that the variable X is bounded by a type where X occurs. We usually encounter F-bounded quantification with A{X} of the form G(X) (for example, $Max_{Op}(X)$).

4.3 Higher-Order Subtyping

A second common property of Max and MinMax can be brought to light by defining a pointwise subtype order on type operators:

$$F \prec: G \qquad \text{if, for all } X, \quad F(X) <: G(X)$$

Just like type operators correspond to protocols, the subtype order on type operators is a direct formalization of the notion of protocol extension.

With this definition, we obtain:

$$\mathrm{Max}_{\mathrm{Op}} \prec: \mathrm{Max}_{\mathrm{Op}}$$
$$\mathrm{MinMax}_{\mathrm{Op}} \prec: \mathrm{Max}_{\mathrm{Op}}$$

Hence, we may want to parameterize over all type operators X with the property that $X \prec: \mathrm{Max}_{\mathrm{Op}}$. A polymorphic function of type:

$$\forall(X \prec: \mathrm{Max}_{\mathrm{Op}})B\{X\}$$

can be instantiated, by application to the type operators $\mathrm{Max}_{\mathrm{Op}}$ and $\mathrm{MinMax}_{\mathrm{Op}}$; the corresponding instances have types $B\{\mathrm{Max}_{\mathrm{Op}}\}$ and $B\{\mathrm{MinMax}_{\mathrm{Op}}\}$. We need to be careful about how X is used in B{X}, because X is now a type operator. Our general strategy is to take the fixpoint of X whenever required; for example, we can obtain a typing for pre-max, as follows:

$$\begin{array}{l} \text{pre-max} : \forall(X \prec: \mathrm{Max}_{\mathrm{Op}})X^* \to X^* \to X^* \quad \triangleq \\ \quad \lambda(X \prec: \mathrm{Max}_{\mathrm{Op}})\ \lambda(\mathrm{self}:X^*)\ \lambda(\mathrm{other}:X^*) \\ \quad\quad \text{if self.n>other.n then self else other} \end{array}$$

$$\text{pre-max}(\mathrm{MinMax}_{\mathrm{Op}}) : \mathrm{MinMax} \to \mathrm{MinMax} \to \mathrm{MinMax}$$

The code of pre-max is well-typed because $X \prec: \mathrm{Max}_{\mathrm{Op}}$ implies $X^* = X(X^*) <: \mathrm{Max}_{\mathrm{Op}}(X^*)$ by definition of $\prec:$, and self:X^* implies self : $\mathrm{Max}_{\mathrm{Op}}(X^*) = [n:\mathrm{Int}, \max^+:X^* \to X^*]$ by subsumption; similarly, other : $[n:\mathrm{Int}, \max^+:X^* \to X^*]$.

In this derivation we have used the unfolding property $X^*=X(X^*)$. An alternative, that greatly simplifies the treatment of higher-order theories, is to assume only an explicit isomorphism between each recursive type and its unfolding, with no equality between the types. Two term-level primitives, fold and unfold, realize the isomorphism:

$$\mathrm{fold} : A\{\mu(X)A\{X\}\} \to \mu(X)A\{X\}$$
$$\mathrm{unfold} : \mu(X)A\{X\} \to A\{\mu(X)A\{X\}\}$$

When we abandon the equality $X^*=X(X^*)$, we need to modify the code of pre-max; it suffices to write unfold(self).n and unfold(other).n instead of self.n and other.n.

4.4 Semantic Equivalence

We have encountered types of the forms $\forall(X<:D(X))B\{X\}$ and $\forall(Y \prec: D)B\{Y^*\}$, where D is a type operator and where if X is a type then B{X} is a type. We sketch an informal argument that $\forall(X<:D(X))B\{X\}$ and $\forall(Y \prec: D)B\{Y^*\}$ are semantically equal.

Let us consider a model of types in terms of sets of some sort, such as an ideal model or a per model [7, 8, 12, 17, 20, 32]. In these models, the interpretation of a type is a set, and the interpretation of a type operator is a function on sets. Quantification is interpreted by intersection. Type recursion is interpreted by unique fixpoints of contractive operators.

Let S, F, and G be the interpretations of A, $\lambda(X)B\{X\}$, and D, respectively. Then, $\bigcap_{X\subseteq S} F(X)$ is the interpretation of $\forall(X<:A)B\{X\}$, and fix(F) is the interpretation of $\mu(X)B\{X\}$. Moreover, let $G\preccurlyeq F$ mean that F and G are contractive and that, for all T, $G(T) \subseteq F(T)$ [13]; the interpretation of $\forall(X<:D)B\{X\}$ is $\bigcap_{U\preccurlyeq G} F(U)$. The contractiveness requirement for $G\preccurlyeq F$ is motivated by types of the form $\forall(Y<:D)...Y^*...$: the interpretation of Y^* requires Y to be contractive; additionally we assume that D is syntactically restricted so that its interpretation is a contractive operator. In particular, $\lambda(X)[v_i{:}B_i{}^{i\in I}, m_j{}^+{:}C_j\{X\}{}^{j\in J}]$ is a suitable D.

Under these assumptions, we can now prove that $\forall(X<:D(X))B\{X\}$ and $\forall(Y<:D)B\{Y^*\}$ are semantically equal, that is, that $\bigcap_{X\subseteq G(X)} F(X)$ and $\bigcap_{U\preccurlyeq G} F(\text{fix } U)$ are equal sets:

- First, $\bigcap_{U\preccurlyeq G} F(\text{fix } U)$ is included in $\bigcap_{X\subseteq G(X)} F(X)$: If $T\subseteq G(T)$ then there exists H such that $H\preccurlyeq G$ and (fix H) = T. An appropriate choice is H = $\lambda(X)(G(X)\cap T)$. We have that $H\preccurlyeq G$ because $H(X)\subseteq G(X)$ for all X and because contractiveness is preserved by intersection. In addition, $H(T) = G(T)\cap T = T$, since $T\subseteq G(T)$, so T is the fixpoint of H.
- Conversely, $\bigcap_{U\preccurlyeq G} F(\text{fix } U)$ includes $\bigcap_{X\subseteq G(X)} F(X)$: It suffices to notice that if $H\preccurlyeq G$ then (fix H) = H(fix H) $\subseteq$ G(fix H). Therefore, given $H\preccurlyeq G$, there exists $T\subseteq G(T)$ such that T = (fix H), as needed.

The proof of equality uses a binary intersection operation on types. This operation is available in the models of interest, but is not always available syntactically. Therefore, the equality may not apply to particular syntactic systems.

Despite the equality, the higher-order framework is semantically more expressive than the F-bounded framework. The type $\forall(Y<:D)\forall(Z<:Y)B'\{Y^*,Z^*\}$ is not of the form $\forall(Y<:D)B\{Y^*\}$, because of the plain occurrence of Y in the bound Z<:Y. This type has no direct analogue in terms of F-bounded quantification.

4.5 Discussion

F-bounded subtyping and higher-order subtyping are both formal counterparts of protocol extension. These two kinds of subtyping have equal expressive power in many examples; this fact is consistent with the semantic equality established in section 4.4. F-bounded subtyping is appealing because it is a simple extension of second-order subtyping. Higher-order subtyping is appealing because it makes clear that protocols have to do with type operators.

The idea of using higher-order subtyping instead of F-bounded subtyping seems to have occurred to a number of people, including the Abel group at HP (Canning, Cook, Hill, and Olthoff), John Mitchell, and later Luca Cardelli. Although the idea dates back to at least 1989, apparently it was never written down, except in e-mail exchanges. The equality of section 4.4, due to Martín

Abadi, appears in a 1990 e-mail exchange.

In the next two sections, we will see how F-bounded subtyping and higher-order subtyping can be used to interpret the matching relation. F-bounded subtyping was the original intended interpretation of matching [10]. Higher-order subtyping is our new interpretation. It is important to ask whether the two interpretations give rise to the same rules. As we will see, they do not. Higher-order subtyping has some definite advantages.

5 Matching as F-bounded Subtyping

Starting in this section, we consider translating a language with a primitive matching relation to a language without it. We refer to the former as the *source* language and to the latter as the *target* language. In this section, the target language is based on F-bounded subtyping.

5.1 The F-bounded Interpretation

The properties of F-bounded subtyping discussed in section 4.2 suggest that we may interpret matching in terms of F-bounded subtyping. To be precise, we would need to write a formal translation. The central idea of the translation is however rather simple; it is expressed by the following two clauses:

$$\begin{array}{lcl} A <\# B & \approx & A <: B_{Op}(A) \\ \forall(X<\#A)B\{X\} & \approx & \forall(X<:A_{Op}(X))B\{X\} \end{array}$$

Since matching is defined on object types of the source language, we must understand the type variable X as ranging only over those types, and not over arbitrary types of the source language. The typing rules of the source language can enforce this constraint, provided there is a separate category of object types, and a way for asserting that a type belongs to this category. An important rule of the source language would say that, under the assumption $X <\# A$, the type variable X is an object type.

This translation is not defined when the right-hand side of <# is a variable, as in the case of cascading quantifiers:

$$\forall(X<\#A)\ \forall(Y<\#X)\ ... \quad \approx \quad ?$$

An expression of the form $\forall(X<:A_{Op}(X))\ \forall(Y<:X_{Op}(Y))$ ... does not make sense as the translation of this type, because X_{Op} is not defined. Hence, we must adopt the restriction that, whenever $\forall(X<\#A)B$ is a type of the source language, the type A must have the form $\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j^+{:}C_j\{X\}\ ^{j\in J}]$. Therefore, the type structure supported by this translation is somewhat irregular: type variables are not allowed in places where object types are allowed.

Next we examine other aspects of the F-bounded interpretation, finding several shortcomings.

5.2 Reflexivity and Transitivity

If A is an object type of the source language, then we would expect that A <# A. This reflexivity rule is necessary for justifying the instantiation f(A) of a polymorphic function $f : \forall(X{<\#}A)B$. According to the F-bounded interpretation, whenever A_{Op} is defined, we obtain:

$$A <\# A \qquad \approx \quad A <: A_{Op}(A)$$

because $A = A_{Op}(A)$ by the unfolding property of recursive types. However, if A is a type variable X, then X_{Op} is not defined, so $X <: X_{Op}(X)$ does not make sense. Hence, reflexivity does not hold in general.

Turning now to transitivity, if A, B, and C are object types of the source language, then we would expect that A <# B and B <# C imply A <# C; this would mean:

$$A <: B_{Op}(A) \quad \text{and} \quad B <: C_{Op}(B) \quad \text{imply} \quad A <: C_{Op}(A)$$

As in the case of reflexivity, we run into difficulties with type variables. Moreover, we can see that transitivity fails even for closed types, with the following counterexample:

$$\begin{array}{lll} A & \triangleq & \mu(X)[p^+: X\to Int,\ q: Int] \\ B & \triangleq & \mu(X)[p^+: X\to Int] \\ C & \triangleq & \mu(X)[p^+: B\to Int] \end{array}$$

We have both A <# B and B <# C, but we do not have A <# C (because $[p^+{:}A\to Int,\ q{:}Int] <: [p^+{:}B\to Int]$ fails).

We can trace this problem back to the definition of D_{Op}, which depends on the exact syntax of the type D. Because of the syntactic character of that definition, two equal types may behave differently with respect to matching. In our example, we have B = C by the unfolding property of recursive types (which is important for F-bounded quantification, as we discussed in section 4.2). Despite the equality B = C, we have A <# B but not A <# C.

5.3 Matching Self

According to the F-bounded interpretation, two types that look rather different may match. Consider two types A and A′ such that:

$$\begin{array}{l} A \equiv \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}] \\ \qquad <\#\ \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j'\{X\}\ ^{j\in J'}] \equiv A' \end{array}$$

This holds when $A <: A'_{Op}(A)$, that is, when $[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{A\}\ ^{j\in J}] <: [v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j'\{A\}\ ^{j\in J'}]$. It suffices that, for every $j\in J'$:

$$C_j\{A\} <: C_j'\{A\}$$

For example, we have:

$$\mu(X)[v{:}Int,\ m^+{:}X] <\# \mu(X)[m^+{:}\ [v{:}Int]]$$

Intuitively, the variable X on the left matches the type [v:Int] on the right. Since X is the Self variable, we may say that Self matches not only Self but also other types (here [v:Int]). This treatment of Self is both sound and flexible. On the other hand, it can be difficult for a programmer to see whether two types match.

6 Matching as Higher-Order Subtyping

In section 5 we discussed some shortcomings of the F-bounded interpretation. In this section we argue that, in contrast, the interpretation of matching as higher-order subtyping is satisfactory, and specifically that it does not share those shortcomings.

6.1 The Higher-Order Interpretation

The higher-order interpretation relies on two translations of types of the source language. In certain contexts, types in the source language are interpreted as types of the target language, while in other contexts they are interpreted as type operators of the target language.

These two translations, $\mathit{Type}\langle A\rangle$ and $\mathit{Oper}\langle A\rangle$, can be informally summarized as follows. For object types of the source language, we set:

$$\mathit{Oper}\langle X\rangle \approx \qquad \text{(assuming that X is match-bound)}$$
$$X$$

$$\mathit{Oper}\langle \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}]\rangle \approx$$
$$\lambda(X)[v_i{:}\mathit{Type}\langle B_i\rangle\ ^{i\in I},\ m_j^+{:}\mathit{Type}\langle C_j\{X\}\rangle\ ^{j\in J}]$$

$$\mathit{Type}\langle X\rangle \approx \qquad \text{(when X is match-bound)}$$
$$X^*$$

$$\mathit{Type}\langle \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}]\rangle \approx$$
$$\mu(X)[v_i{:}\mathit{Type}\langle B_i\rangle\ ^{i\in I},\ m_j^+{:}\mathit{Type}\langle C_j\{X\}\rangle\ ^{j\in J}]$$

For other types, we set:

$$\mathit{Type}\langle X\rangle \approx X \quad \text{(when X is not match-bound)}$$
$$\mathit{Type}\langle A\to B\rangle \approx \mathit{Type}\langle A\rangle\to\mathit{Type}\langle B\rangle$$
$$\mathit{Type}\langle \forall(X<\#A)B\rangle \approx \forall(X<:\mathit{Oper}\langle A\rangle)\mathit{Type}\langle B\rangle$$

Here we assume the restriction that, whenever $\forall(X<\#A)B$ is a type of the source language, A must be an object type, that is, either a match-bound variable or a type of the form $\mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}]$. For example, $\mu(X)[l{:}\forall(Y<\#X)B, \dots]$ is not allowed, while $\forall(X<\#A)\ [l{:}\forall(Y<\#X)B, \dots]$ is.

When $X <\# A$ is the bound of a quantifier, we distinguish the occurrences of X in a type context (as in $X \rightarrow X$) from the ones in an operator context (as in $\forall(Y{<}\#X)...$). The bound is translated as $X \prec: A_{Op}$; the occurrences of X in a type context are translated as X^*, while the occurrences in an operator context are translated simply as X. For instance:

$$\begin{array}{l} \mathit{Type}\langle \forall(X{<}\#Max)\ \forall(Y{<}\#X)\ X \rightarrow Y \rangle \ \approx \\ \quad \forall(X \prec: Max_{Op})\ \forall(Y \prec: X)\ X^* \rightarrow Y^* \end{array}$$

This translation is well-defined on type variables, so now there are no problems with cascading quantifiers.

Given these two translations of types, the translation of matching is now:

$$A <\# B \quad \approx \quad \mathit{Oper}\langle A \rangle \prec: \mathit{Oper}\langle B \rangle \qquad \text{for A, B object types}$$

The higher-order interpretation does not use the unfolding property of recursive types for the *target* language; instead, it uses explicit fold and unfold primitives as explained in section 4.3. On the other hand, the higher-order interpretation is incompatible with the unfolding property of recursive types in the *source* language, because $\mathit{Oper}\langle \mu(X)A\{X\} \rangle$ and $\mathit{Oper}\langle A\{\mu(X)A\{X\}\} \rangle$ are in general different type operators. The unfolding property of recursive types is convenient for programming, but it is not an essential feature and it is the origin of technical complications. We are fortunate to be able to drop it throughout.

6.2 Reflexivity and Transitivity

Reflexivity is now satisfied by all object types, including variables; for every object type A, we have:

$$A <\# A \quad \approx \quad \mathit{Oper}\langle A \rangle \prec: \mathit{Oper}\langle A \rangle$$

This follows from the reflexivity of $\prec:$, given that $\mathit{Oper}\langle A \rangle$ is a well-defined type operator for every object type A of the source language.

Similarly, transitivity is satisfied by all triples A, B, and C of object types, including variables:

$$\begin{array}{l} A <\# B \ \text{ and } \ B <\# C \ \text{ imply } \ A <\# C \ \approx \\ \quad \mathit{Oper}\langle A \rangle \prec: \mathit{Oper}\langle B \rangle \ \text{ and } \ \mathit{Oper}\langle B \rangle \prec: \mathit{Oper}\langle C \rangle \\ \qquad \text{imply } \ \mathit{Oper}\langle A \rangle \prec: \mathit{Oper}\langle C \rangle \end{array}$$

This follows from the transitivity of $\prec:$. The counterexample of section 5.2 does not apply, because there $B <\# C$ does not hold under the higher-order interpretation.

6.3 Matching Self

With the higher-order interpretation, the relation:

$$A \equiv \mu(\text{Self})[v_i{:}B_i\ ^{i\in I},\ m_j^{+}{:}C_j\{\text{Self}\}\ ^{j\in J}]$$
$$\quad <\#\ \mu(\text{Self})[v_i{:}B_i\ ^{i\in I},\ m_j^{+}{:}C_j'\{\text{Self}\}\ ^{j\in J'}] \equiv A'$$

holds when the type operators corresponding to A and A′ are in the subtyping relation, that is, when:

$$[v_i{:}\mathit{Type}\langle B_i\rangle\ ^{i\in I},\ m_j^{+}{:}\mathit{Type}\langle C_j\{\text{Self}\}\rangle\ ^{j\in J}]$$
$$\quad <:\ [v_i{:}\mathit{Type}\langle B_i\rangle\ ^{i\in I},\ m_j^{+}{:}\mathit{Type}\langle C_j'\{\text{Self}\}\rangle\ ^{j\in J'}] \qquad \text{for an arbitrary Self}$$

For this, it suffices that, for every j in J′:

$$\mathit{Type}\langle C_j\{\text{Self}\}\rangle\ \ <:\ \ \mathit{Type}\langle C_j'\{\text{Self}\}\rangle$$

Since Self is μ-bound, all the occurrences of Self are translated as Self*. Then, an occurrence of Self* on the left can be matched only by a corresponding occurrence of Self* on the right, since Self is arbitrary. In short, *Self matches only itself*. This property makes it easy for programmers to glance at two object types and tell whether they match. On the other hand, it may be convenient to let Self match more types, as in the F-bounded interpretation, but the higher-order interpretation does not allow it.

7 Inheritance and Classes via Higher-Order Subtyping

It is possible to make rigorous the translation of section 6, thus proving that matching can be explained in terms of higher-order subtyping. We do not do this here, but instead give further evidence by writing some examples and by showing that the basic properties of matching are supported and that binary methods can be inherited. See [6] for a more systematic treatment.

In section 3.1, we listed the following expected typings:

if X<#Inc and x:X then x.n : Int
if X<#Inc and x:X and b:Int then x.n:=b : X

The higher-order interpretation induces the following term translations:

if X≺:Inc_{Op} and x:X* then unfold(x).n : Int
if X≺:Inc_{Op} and x:X* and b:Int then fold(unfold(x).n:=b) : X*

For the first typing, we have unfold(x):X(X*). Moreover, from X≺:Inc_{Op} we obtain X(X*) <: Inc_{Op}(X*) = [n:Int, inc:X*]. Therefore, unfold(x):[n:Int, inc:X*], and unfold(x).n:Int.

For the second typing, we have again unfold(x):X(X*) with X(X*) <: [n:Int, inc:X*]. We then use a typing rule for field update in the target language [5]. This rule says that if a:A, c:C, and A <: [v:C,...] then (a.v:=c) : A. In our case, we have unfold(x):X(X*), b:Int, and X(X*) <: [n:Int, inc:X*]. We obtain (unfold(x).n:=b) : X(X*). Finally, by folding, we obtain fold(unfold(x).n:=b) : X*.

Applying our higher-order translation to MaxClass from section 3.2, we obtain:

$$\begin{aligned}&\text{MaxClass} \triangleq\\&\quad[\text{new}^{+}\text{: Max},\\&\quad\ \text{n: Int},\\&\quad\ \text{max: } \forall(X{<\!\!:}\text{Max}_{Op})X^{*}\to X^{*}\to X^{*}]\end{aligned}$$

The corresponding translation at the term level produces:

```
maxClass : MaxClass  ≜
    [new = ς(classSelf: MaxClass)
            fold(
                [n = classSelf.n,
                 max = ς(self:Max_Op(Max))
                            classSelf.max(Max_Op)(fold(self))]),
     n = 0,
     max = pre-max]
pre-max : ∀(X<:Max_Op)X*→X*→X*  ≜
    λ(X<:Max_Op) λ(self:X*) λ(other:X*)
        if unfold(self).n>unfold(other).n then self else other
```

It is easy to check that pre-max is well typed, similarly to the derivation of the first typing above.

The instantiations pre-max(Max_{Op}) and pre-max(MinMax_{Op}) are both legal. Since pre-max has type $\forall(X{<\!\!:}\text{Max}_{Op})X^{*}\to X^{*}\to X^{*}$, this pre-method can be used as a component of a class of type MaxClass. Moreover, a higher-order version of the rule for quantifier subtyping yields:

$$\forall(X{<\!\!:}\text{Max}_{Op})X^{*}\to X^{*}\to X^{*} \quad <: \quad \forall(X{<\!\!:}\text{MinMax}_{Op})X^{*}\to X^{*}\to X^{*}$$

so pre-max has type $\forall(X{<\!\!:}\text{MinMax}_{Op})X^{*}\to X^{*}\to X^{*}$ by subsumption, and hence pre-max can be reused as a component of a class of type MinMaxClass.

8 Conclusions

As we have seen, there are situations in programming where one would like to parameterize over all "extensions" of a recursive object type, rather than over all its subtypes. Both F-bounded subtyping and higher-order subtyping can be used for capturing the notion of extension; however, they are probably too complex for overt use in programming. Using a primitive matching relation is more appropriate. Thus, the concept of matching represents a step forward towards practical and flexible type theories for object-oriented programming.

Still, F-bounded subtyping and higher-order subtyping have an important role to play in explaining the semantics of matching and in justifying its rules.

We have presented two interpretations of matching:

$$A <\# B \quad \approx \quad A <: B_{Op}(A) \qquad \text{(F-bounded interpretation)}$$

$$A <\# B \quad \approx \quad A_{Op} \prec: B_{Op} \qquad \text{(higher-order interpretation)}$$

As these interpretations show, matching stems from subtyping. The former interpretation motivated the concept of matching, and led to the typing rules of TOOPLE, which are based directly on the F-bounded interpretation. The latter interpretation leads to rules that are much closer to the ones of PolyTOIL.

Although both interpretations can be soundly adopted, they require different assumptions and yield different rules. In particular, the higher-order interpretation does not assume the unfolding property of recursive types, which seems necessary for the F-bounded interpretation; this simplification is technically advantageous in a higher-order setting. Additionally, the higher-order interpretation validates reflexivity and transitivity properties for matching; these properties do not hold under the F-bounded interpretation. Thus, we believe that the higher-order interpretation is preferable; it should be a guiding principle for programming languages with matching.

Acknowledgments

We thank Kim Bruce, Benjamin Pierce, and Scott Smith for useful discussions.

References

[1] Abadi, M., **Baby Modula-3 and a theory of objects**. *Journal of Functional Programming* **4**(2), 249-283. 1994.

[2] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag. 1994.

[3] Abadi, M. and L. Cardelli, **A semantics of object types**. *Proc. IEEE Symposium on Logic in Computer Science*, 332-341. 1994.

[4] Abadi, M. and L. Cardelli, **A theory of primitive objects: second-order systems**. *Proc. ESOP'94 - European Symposium on Programming*. Springer-Verlag. 1994.

[5] Abadi, M. and L. Cardelli, **An imperative object calculus**. *TAPSOFT'95: Theory and Practice of Software Development*. P.D. Mosses, M.Nielsen, and M.I.Schwartzbach Eds., Lecture Notes in Computer Science 915, 471-485. Springer-Verlag. 1995.

[6] Abadi, M. and L. Cardelli, **A theory of objects**. Springer-Verlag. (*To appear*.) 1996.

[7] Abadi, M. and G. Plotkin, **A per model of polymorphism and recursive types**. *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, 355-365. 1990.

[8] Amadio, R.M., **Recursion over realizability structures**. *Information and Computation* **91**(1), 55-85. 1991.

[9] Amadio, R.M. and L. Cardelli, **Subtyping recursive types**. *ACM Transactions on Programming Languages and Systems* **15**(4), 575-631. 1993.

[10] Bruce, K.B., **A paradigmatic object-oriented programming language: design, static typing and semantics**. *Journal of Functional Programming* **4**(2), 127-206. 1994.

[11] Bruce, K.B., L. Cardelli, G. Castagna, The Hopkins Objects Group, G.T. Leavens, B. Pierce, **On binary methods**. *To appear in TAPOS.*

[12] Bruce, K.B. and G. Longo, **A modest model of records, inheritance and bounded quantification**. *Information and Computation* **87**(1/2), 196-240. 1990.

[13] Bruce, K.B. and J.C. Mitchell, **PER models of subtyping, recursive types and higher-order polymorphism**. *Proc. 19nd Annual ACM Symposium on Principles of Programming Languages*, 316-327. 1992.

[14] Bruce, K.B., A. Schuett, and R. van Gent, **PolyTOIL: a type-safe polymorphic object-oriented language.** *Proc. ECOOP'95.* Springer-Verlag. 1995.

[15] Canning, P., W. Cook, W. Hill, W. Olthoff, and J.C. Mitchell, **F-bounded polymorphism for object-oriented programming**. *Proc. ACM Conference on Functional Programming and Computer Architecture*, 273-280. 1989.

[16] Cardelli, L., **A semantics of multiple inheritance**. *Information and Computation* **76**, 138-164. 1988.

[17] Cardelli, L. and G. Longo, **A semantic basis for Quest**. *Journal of Functional Programming* **1**(4), 417-458. 1991.

[18] Cardelli, L. and J.C. Mitchell, **Operations on records**. *Mathematical Structures in Computer Science* **1**(1), 3-48. 1991.

[19] Cardelli, L., J.C. Mitchell, S. Martini, and A. Scedrov, **An extension of system F with subtyping**. *Information and Computation* **109**(1-2), 4-56. 1994.

[20] Cardone, F., **Relational semantics for recursive types and bounded quantification**. *Proc. Automata, Languages and Programming*, 164-178. Lecture Notes in Computer Science 372. Springer-Verlag. 1989.

[21] Compagnoni, A.B., **Higher-order subtyping with intersection types**. *Ph.D. Thesis,* Cip-Data Koninklijke Bibliotheek, Den Haag, Nijmegen. 1995.

[22] Cook, W., W. Hill, and P. Canning, **Inheritance is not subtyping**. *Proc. Seventeenth Annual ACM Symposium on Principles of Programming Languages*. 1990.

[23] Curien, P.-L. and G. Ghelli, **Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$**. *Mathematical Structures in Computer Science* **2**(1), 55-91. 1992.

[24] Danforth, S. and C. Tomlinson, **Type theories and object-oriented programming**. *ACM Computing Surveys* **20**(1), 29-72. 1988.

[25] Day, M., R. Gruber, B. Liskov, and A.C. Myers, **Subtypes vs. where clauses: constraining parametric polymorphism**. P*roc. OOPSLA'95*, 156-168. 1995.

[26] Eifrig, J., S. Smith, V. Trifonov, and A. Zwarico, **An interpretation of typed OOP in a language with state**. *Proc. OOPSLA'94*, 16-30. 1994.

[27] Girard, J.-Y., **Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur**. Thèse de doctorat d'état, University of Paris. 1972.

[28] Gunter, C.A. and J.C. Mitchell, ed., **Theoretical Aspects of Object-Oriented Programming**. MIT Press. 1994.

[29] Harper, R. and B. Pierce, **A record calculus based on symmetric concatenation**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*. 1991.

[30] Hofmann, M. and B.C. Pierce, **A unifying type-theoretic framework for objects**. *Proc. Symposium on Theoretical Aspects of Computer Science*. 1994.

[31] Katiyar, D., D. Luckham, and J.C. Mitchell, **Polymorphism and subtyping in interfaces**. *ACM SIGPLAN Notices* **29**(8), 22-34. 1994.

[32] MacQueen, D.B., G.D. Plotkin, and R. Sethi, **An ideal model for recursive poly-**

morphic types. *Information and Control* **71**, 95-130. 1986.

[33] Mitchell, J.C., **Toward a typed foundation for method specialization and inheritance**. *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, 109-124. 1990.

[34] Nelson, G., ed. **Systems programming with Modula-3**. Prentice Hall. 1991.

[35] Steffen, M. and B.C. Pierce, **Higher-order subtyping**. *Technical Report ECS-LFCS-94-280*. University of Edinburgh. 1994.