An Abstract View of Objects and Subtyping

(Preliminary Report)

Martin Hofmann **Benjamin Pierce**

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh, EH9 3JZ, U.K.

August 31, 1992
ECS-LFCS-92-226

Abstract

We give a direct type-theoretic characterization of the basic mechanisms of object-oriented programming — objects, methods, message passing, and subtyping — by introducing an explicit *Object* type constructor together with suitable introduction, elimination, and equality rules. The resulting abstract framework represents a first step towards a high-level axiomatization of object-oriented programming and provides a common basis for justifying and comparing known encodings of objects in terms of the seemingly diverse machinery of recursive record types [9, 12], F-bounded quantification [6, 17], and existential types [33, 32].

1 Introduction

An *object* can be described as a *state component* of some hidden *representation type* and a collection of *methods* that can be used to analyze or modify the state. For example, consider a collection of simple one-dimensional point objects, each supporting the operations `setX` and `getX`, where `setX` is expressed in a functional style — returning an object with an updated state component rather than modifying it in-place. Such objects can be implemented by choosing a representation type, say `Int`, a state component, say `5`, and two method implementations, say

```
setX = fun(state:Int) fun(newX:Int) newX
getX = fun(state:Int) state
```

and packaging them together so that the state component is protected from external access except through the method implementations.

Unlike the elements of ordinary abstract data types, different point objects may have different internal representations; every point comes with its own implementation of the `setX` and `getX` methods, appropriate to its internal representation type. Thus, another implementation of point objects might use the representation type `{x:Int,other:Int}`, the initial state `{x=5,other=8}`, and the methods:

```
setX = fun(state:{x:Int,other:Int}) fun(i:Int) {x=i, other=state.other}
getX = fun(state:(x:Int,other:Int}) state.x
```

This flexibility is crucial to the spirit of object-oriented programming. Although it is not present in precisely this form in some object-oriented languages (e.g. Smalltalk [22] and C++ [39]), equivalent mechanisms like "virtual classes" are invariably introduced in its place. When a message is sent to an object, the identity of the object itself determines what code is executed in response.

The variability in implementations implies that a program manipulating a point object must do so "generically" — by calling the point's methods to analyze and update its state as necessary — rather than concretely, by direct operations on the state. In other words, it should be implemented in terms of a *uniform* function

```
Point'setX : Point -> Int -> Point
```

that, given a point, invokes its internal `setX` method and packages the resulting concrete representation into a new instance of `Point`; and another function

```
Point'getX : Point -> Int
```

that uniformly invokes the `getX` method of any point and returns the resulting integer.

Our purpose here is to study these central features, encapsulation and message passing (and, in less detail, the related notion of subtyping), in a type-theoretic setting. Throughout the paper, we shall have two parallel goals:

1. To propose an abstract type-theoretic characterization of objects and message passing, a first step towards a high-level framework for reasoning about typed object-oriented programs.
2. To use this framework to eludicate the relationship between previously studied type-theoretic encodings of objects in terms of existential types and recursive types.

In Section 2, we introduce the basic constructions of our abstract framework. We state a simple syntactic condition on object types, capturing the intuition that methods can only access the state of one object at a time, and show that this yields a natural definition of uniform method invocation. In Section 3, we extend this framework to include subtyping. In Section 4 we use it to define a high-level syntax for object type declarations and message passing. Section 5 shows that a simple encoding of objects in terms of existential types satisfies our axioms. Section 6 further justifies our framework by showing that the more familiar encoding of object types as recursive records satisfies the axioms as well; Section 7 discusses the special case of F-bounded quantification.

We close with two tentative extensions. Section 8 shows how the object axioms in Section 2 can be justified in terms of more fundamental mathematical considerations, using an argument based on parametricity. Sections 9 and 10 extend the framework to mixed-variance method signatures, illustrating the correspondence between our approach and previously studied encodings of mixed-variance objects. In the mixed-variance case, the extra power of recursive types gives rise to a more satisfactory encoding, since it correctly captures subtyping while the encoding in terms of existential types does not.

2 Objects

Since the representation type of an object is hidden from external view, its *interface* is just the types of its methods. Type-theoretically, this can be modeled as a type operator with one argument, thought of as a type with one free variable that stands for the hidden representation type. For example, the interface of point objects is described by the operator

```
PointM = Fun(X) {setX: X->Int->X, getX: X->Int}.
```

For now, we leave unspecified the "ambient type theory" in which the definitions in this section are embedded. We will be using various extensions of F^{ω} [21], summarized in Appendix A. The calculus under consideration will determine the precise force of the equational constraints expressed by the diagrams in this section.

To characterize the set of objects sharing a common interface without committing ourselves to any particular encoding of objects, we introduce a new type constructor *Object*, which turns an interface specification into a type:

$$\frac{\Gamma \vdash M \in \star\!\to\!\star}{\Gamma \vdash \mathit{Object}(M) \in \star}$$

So the type of point objects is

```
Point = Object(PointM).
```

Elements of an object type are created using the term constructor *object*. Given an interface specification M, a concrete representation type R, a collection m of concrete methods, and an initial value s of the representation type, we use *object* to package them together into an element $\mathit{object}_M(R, s, m) \in \mathit{Object}(M)$. For example, two point objects with different representation types can be created as follows:

```
m1 = {setX = fun(state:Int) fun(i:Int) i,
      getX = fun(state:Int) state}
   : PointM(Int)
p1 = object_PointM(Int, 5, m1)
   : Point

m2 = {setX = fun(state:{x:Int,other:Int})
                 fun(i:Int)
                   {x=i, other=state.other},
      getX = fun(state:(x:Int,other:Int}) state.x}
   : PointM({x:Int,other:Int})
p2 = object_PointM({x:Int,other:Int}, {x=5,other=8}, m2)
   : Point
```

The *object* constructor has the following typing rule:

$$\frac{\Gamma \vdash M \in \star\!\rightarrow\!\star \qquad \Gamma \vdash s \in R \qquad \Gamma \vdash m \in M(R)}{\Gamma \vdash \mathit{object}_M(R, s, m) \in \mathit{Object}(M)}$$

(Later, it will turn out to be convenient to alter this rule slightly. Final versions of all the rules are given in Appendix B.)

For the elimination of elements of object types, we might use an unpacking rule in the style of the existential elimination rule of Mitchell and Plotkin [28]. But this runs counter to the spirit of object-style programming, in which objects are never "opened" but are acted on externally by sending *messages* to invoke their internal methods. We intend to capture this mechanism directly.

Since every point object must implement the `setX` method, we want to have a uniform function

```
Point'setX : Point -> Int -> Point
```

that, given a point, invokes its `setX` method. More generally, for each operator M representing the interface of an object type, we introduce a term constant GM_M denoting the "generic method" for objects with interface M:

$$\frac{\Gamma \vdash M \in \star\!\rightarrow\!\star}{\Gamma \vdash GM_M \in M(\mathit{Object}(M))}$$

Then

```
GM_Point : {setX: Point->Int->Point, getX: Point->Int}
```

and

```
Point'setX = GM_Point.setX.
```

But such uniform functions do not necessarily exist. For example, we can extend our first implementation of points with an equality method

```
eq = fun(state1:Int) fun(state2:Int) (state1 = state2)
```

but we cannot expect to be able to invoke this `eq` uniformly — that is, we cannot expect to have a function

```
Point'eq : Point -> Point -> Bool
```

that calls the `eq` method of its first parameter and passes it the internal representation of the second parameter: such an invocation of the low-level `eq` function will only be well typed when the two points passed as arguments to `Point'eq` happen to have *identical* representation types, which `Point'eq` is clearly unable to check.[1] This does not mean, of course, that equality comparisons between points cannot be implemented at all. We can write an external equality *function* (as opposed to an equality *method* local to one of the points) using the `getX` method of both points generically, as

[1] Unable, at any rate, in a language with a parametric semantic model [37]. Some object-oriented languages do include non-parametric features such as run-time interrogation of the "class" of an object [22, 23], which can support methods that operate on the concrete representation of more than one object at a time. We restrict our attention here to the parametric case.

```
eqPoint = fun(p,q:Point) eqInt (Point'getX p) (Point'getX q)
```

where `eqInt` ∈ `Int->Int->Bool` is the equality function on integers. This point of view is developed in more detail in [31].

The difficulty illustrated by this example corresponds to a well-known limitation of object-style encapsulation [36, 16]. In most object-oriented languages, there is no way to write a method that has concrete access to the internal state of more than one object simultaneously. This limitation can be modeled abstractly as a syntactic restriction on the form of M, capturing the intuition that the methods M describes should all be "essentially unary functions" of the representation type. A "unary" operator M is one of the form

$$M(X) = \{l_1{:}X{\rightarrow}N_1(X),\ \ldots, l_n : X{\rightarrow}N_n(X)\}$$

or, more simply,

$$M(X) = X \rightarrow N(X)$$

for $N(X) = \{l_1{:}N_1(X),\ \ldots, l_n : N_n(X)\}$, where N_1 through N_n contain X only in positive positions. (A variable appears only positively in a type if every occurrence is on the left hand side of an even number of arrows. For polymorphic types, recursive types, and type operators, some additional considerations apply. The notion of positive occurrence is defined in Appendix A.7 by induction on the structure of types.) We write $co(\mathit{Fun}(A)\,T)$ in formulas to assert that the parameter A only appears positively in the body T.

We will henceforth assume that every M under consideration has this well-behaved form. To simplify the development, we take N rather than M as the interface of an object type, so that, for example,

```
Point = Object(PointN),
```

where

```
PointN = Fun(X) {setX: Int->X, getX: Int}.
```

This means that we need to change the typing rules for the *object* constructor and the generic method:[2]

$$\frac{\Gamma \vdash N \in \star{\rightarrow}\star \qquad co(N) \qquad \Gamma \vdash s \in R \qquad \Gamma \vdash m \in R{\rightarrow}N(R)}{\Gamma \vdash \mathit{object}_N(R, s, m) \in \mathit{Object}(N)}$$

$$\frac{\Gamma \vdash N \in \star{\rightarrow}\star \qquad co(N)}{\Gamma \vdash GM_N \in \mathit{Object}(N) \rightarrow N(\mathit{Object}(N))}$$

Next, we must consider a suitable axiomatization of the behavior of generic methods. This should reflect the intuition that a generic method should be a "packaged version of" the method that was originally used to build an object. More precisely, the result

[2] It would be formally cleaner here to allow the formation of $\mathit{Object}(N)$ only when N is functorial, and to state the other rules only for well-formed object types. But stating the rules in this way would require, in Section 3, that we be able to quantify over covariant operators. The study of a refined type theory in which this would be possible is an interesting topic in its own right, but goes beyond the scope of this paper.

of applying the generic method to an initial state packed up into an object should be the same as the result of applying the original method to the original state and then repacking the result — i.e., the informal diagram

$$\begin{tikzcd} \mathit{Object}(N) \arrow[r, "GM_N"] & N(\mathit{Object}(N)) \\ R \arrow[u, "\mathit{object}_N\,(R,\,—,\,m)"] \arrow[r, "m"'] & N(R) \arrow[u, "\text{“repack”}"'] \end{tikzcd}$$

should commute, where the "dash-expression" $\mathit{object}_N\,(R, —, m)$ stands for the packing function $\mathit{fun}(x : R)\ \ \mathit{object}_N\,(R, x, m)$.

In the case of points, this corresponds to the equations

```
  (GM_PointN (object_PointN(Int, s, m1))).setX(i)
= object_PointN(Int, (m1 s).setX(i), m1)
```

and

```
  (GM_PointN (object_PointN(Int, s, m1))).getX
= s,
```

where

```
m1 = fun(s:Int) {setX = fun(i:Int) i, getX = s}.
```

So the arrow labeled "repack" in this case is the function

```
fun(r: {setX:Int->Int,getX:Int})
  {setX = fun(i:Int) object_PointN(Int, r.setX i, m1),
   getX = r.getX}.
```

For the general case, the repacking function can be defined using the more fundamental notion of functorial strength.

2.1. Definition: A type operator $N \in \star\to\star$ is *functorial* if there exists a function

$$\mathit{strength}_N \in \mathit{All}(X)\,\mathit{All}(Y)\ \ (X\to Y) \to (N(X)\to N(Y))$$

such that if $f \in S\to T$ and $g \in T\to U$, then

$$\mathit{strength}_N\ [S]\ [U]\ (f;g) = \mathit{strength}_N\ [S]\ [T]\ (f)\ ;\ \mathit{strength}_N\ [T]\ [U]\ (g)$$

(where "$f;g = \lambda x.\ g(f(x))$" denotes composition in diagrammatic order) and such that

$$\mathit{strength}_N\ [S]\ [S]\ (\mathit{id}[S]) = \mathit{id}\ [N(S)].$$

For the type systems used in Sections 5 and 6, the action of a positive type operator N on a function $f \in X\to Y$ can be interpreted as "applying f to each occurrence of X in $N(X)$" — that is, given an element $n \in N(X)$ by decomposing n, apply f to each component of type X and use the results to rebuild an element of $N(Y)$. For example, if $N(X) = A + (X \times X)$, this procedure yields

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{case}\ n\ \mathit{of}} \\
\quad\ \ \mathit{inl}(a) & \Rightarrow & \mathit{inl}(a) \\
\quad \mid\ \mathit{inr}(r) & \Rightarrow & \mathit{inr}(f(\mathit{fst}(r)), f(\mathit{snd}(r))) \\
\multicolumn{3}{l}{\in\ A + (Y \times Y).}
\end{array}
$$

The term "strength" is borrowed from category theory [24], where an endofunctor $N \in C \rightarrow C$ on a cartesian closed category C is called "strong" if its action on morphisms can be internalized, i.e., if there exists a natural transformation

$$\mathit{str}_{X,Y} : (X \Rightarrow Y) \rightarrow (N(X) \Rightarrow N(Y))$$

(the "strength" of N) that captures the action of N.

2.2. Fact: In the type theories under consideration, every positive operator is functorial and its strength can be defined by induction on its structure (see Appendix A.7). We will henceforth use the terms "positive," "covariant," and "functorial" interchangeably.

Now we can specify the behavior of generic methods for an arbitrary positive operator N. Given a representation type R, a state $s \in R$, and a concrete method $m \in R \rightarrow N(R)$, we require that the diagram

$$
\begin{tikzcd}
\mathit{Object}(N) \arrow[r, "GM_N"] & N(\mathit{Object}(N)) \\
R \arrow[u, "\mathit{object}_N(R,\, —,\, m)"] \arrow[r, "m"'] & N(R) \arrow[u, "{\mathit{strength}_N\ [R]\ [\mathit{Object}(N)]\ \mathit{object}_N(R,\, —,\, m)}"']
\end{tikzcd}
\qquad (\dagger)
$$

commute; or, equivalently, that the following equation be satisfied:

$$
\begin{array}{ll}
 & GM_N(\mathit{object}_N(R,\ s,\ m)) \\
= & \mathit{strength}_N\ [R]\ [\mathit{Object}(N)]\ (\mathit{object}_N(R,\ —,\ m))\ (m\ s) \\
 & \\
\in & N(\mathit{Object}(N)).
\end{array}
$$

Indeed, if we orient this equation from left to right, we obtain a natural computation rule for objects.

Note that the specification ($\dagger$) constrains only the behavior of those elements of $\mathit{Object}(N)$ that lie in the image of the packing function $\mathit{object}_N(R,\ —,\ m)$. Operationally, this is sufficient, since every object occurring in a program must at some stage have been constructed with this function. However, it may be desirable to internalize this observation by imposing an additional η-like equation:

$$\frac{\Gamma \vdash x \in \mathit{Object}(N)}{\Gamma \vdash x \ =\ \mathit{object}_N(\mathit{Object}(N),\ x,\ GM_N) \in \mathit{Object}(N)} \qquad \text{(Eq-Object-Eta)}$$

We prefer to regard this axiom as optional, since it places a strong constraint on the encodings we discuss below. But we shall prove that under certain uniformity assumptions, (η) is actually implied by (†), up to observational equivalence (Lemma 8.6).

2.3. Remark: It is interesting to note that our abstract specification of objects defines *Object*(N) as a weakly terminal co-algebra for the functor N. Indeed, our encoding of objects using existential types in Section 5 corresponds exactly to the impredicative coding of weakly terminal co-algebras proposed by Wraith [40]. It is worth considering modeling objects by (strongly) terminal co-algebras instead. The introduction rule and generic method would remain unchanged in this case, but we would have, in addition, a stronger proof principle for objects based on bisimulation equivalence.

3 Subtyping

Next, we consider extending our abstract characterization of objects and message passing to include another critical concept from object-oriented programming languages: subtyping. Our conclusions in this section are somewhat tentative, firstly because the formal metatheory (in particular, the equational theory) of the type systems we use is not yet fully understood, and secondly because we are not yet confident that we have arrived at a "canonical" formulation of the laws of subtyping for objects. Nevertheless, the extended framework allows a closer comparison of the object encodings using existential and recursive types, and suggests some promising directions for future research. The issues we must deal with are as follows:

1. The lambda-calculus in which the model is expressed must be extended with subtyping.
2. The order-theoretic behavior of the *Object* type constructor must be specified. For example, if we introduce a type of colored point objects whose behavior includes `setC` and `getC` methods in addition to `setX` and `getX`, then we want to be able to consider every colored point as a point.
3. The generic method of a given object type should be applicable to elements of object types with more demanding specifications. Moreover, the fact that sending a message is a kind of *update operation* must be reflected in the typing of the generic method. For example, the generic method of points should be applicable to colored points, and setting the x coordinate of a colored point must yield a colored point.
4. The equational specification of the generic method must be refined to take its new typing into account.

We consider these issues in order.

Various extensions of F^ω with subtyping have been proposed [11, 7, 30, 32]; we choose the simplest of these [30]. The formulation of this system, called $F^\omega_\leq$, follows the same pattern that Cardelli and Wegner used to obtain $F_\leq$ from the pure polymorphic lambda-calculus [21, 35]; details are given in Appendix A.

- The typing relation $\Gamma \vdash e \in T$ is extended with a subtype relation $\Gamma \vdash S \leq T$ and a rule of subsumption:

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T} \qquad \text{(T-SUBSUMPTION)}$$

- For each kind K, we add a maximal element $\mathit{Top}(K)$.
- Binding occurrences of type variables in quantifiers are decorated with subtyping assumptions. In contexts, assumptions about type variables have the form $A{\leq}T$ instead of $A{:}K$.
- To keep the kind structure as simple as possible, type operators retain the form $\mathit{Fun}\,(A{:}K)\,T$ rather than changing to $\mathit{Fun}\,(A{\leq}S)\,T$. This means that the rule for checking the well-formedness of a type operator cannot simply extend the context with the assumption $A{:}K$, but must use $A{\leq}\,\mathit{Top}(K)$ instead:

$$\frac{\Gamma,\ A{\leq}\,\mathit{Top}(K_1)\ \vdash\ T_2\ \in\ K_2}{\Gamma\ \vdash\ \mathit{Fun}\,(A{:}K_1)\,T_2\ \in\ K_1{\rightarrow}K_2} \tag{K-Arrow-I}$$

- We introduce a "pointwise subtyping" rule for operators:

$$\frac{\Gamma,\ A{\leq}\,\mathit{Top}(K)\ \vdash\ S\ \leq\ T}{\Gamma\ \vdash\ \mathit{Fun}\,(A{:}K)\,S\ \leq\ \mathit{Fun}\,(A{:}K)\,T} \tag{S-Abs}$$

  Intuitively, $\mathit{Fun}\,(A{:}K)\,S$ is a subtype of $\mathit{Fun}\,(A{:}K)\,T$ iff $[U/A]S$ is a subtype of $[U/A]T$ for every $U \in K$.
- Because subtyping of operators is pointwise, we may promote the operator in a type application to any larger operator "in place":

$$\frac{\Gamma\ \vdash\ S\ \leq\ T \qquad \Gamma\ \vdash\ S\ U\ \in\ K}{\Gamma\ \vdash\ S\ U\ \leq\ T\ U} \tag{S-App}$$

  The interesting case is when S is a variable, so that $S\ U$ is not a β-redex while $T\ U$ may be.

Since the specification of the generic method depends on the notion of functorial strength in the ambient lambda-calculus, we also need to consider the interaction between strength and subtyping. A natural requirement is that the two should commute:

$$\frac{\begin{array}{cc} \Gamma\ \vdash\ N'\ \leq\ N \qquad \mathit{co}(N') & \mathit{co}(N) \\ \Gamma\ \vdash\ f\ \in\ X{\rightarrow}Y & \Gamma\ \vdash\ n\ \in\ N'(X) \end{array}}{\begin{array}{l} \Gamma\ \vdash\ \mathit{strength}_{N'}\ [X]\ [Y]\ f\ n \\ \quad = \mathit{strength}_{N}\ [X]\ [Y]\ f\ n \\ \quad \in\ N(Y) \end{array}} \tag{Eq-Str-Nat}$$

i.e.,

$$\begin{tikzcd}
N(X) \arrow[rr, "\mathit{strength}_N\ {[X]}\ {[Y]}\ f"] & & N(Y) \\
N'(X) \arrow[u, "\leq"] \arrow[rr, "\mathit{strength}_{N'}\ {[X]}\ {[Y]}\ f"'] & & N'(Y) \arrow[u, "\leq"']
\end{tikzcd}$$

This says precisely that, for each pair of positive operators N and N' such that $N' \leq N$, the family of coercions $\{[\![N'(S) \leq N(S)]\!] \mid S \in \star\}$ in the model forms a natural transformation.

With these extensions of the base calculus, we are ready to deal with object types. We want the *Object* constructor to be monotone, so that

```
CPoint = Object(CPointN) < Object(PointN) = Point,
```

where

```
CPointN = Fun(X) {setX: Int->X, getX: Int,
                  setC: Color->X, getC: Color},
```

so that

```
CPointN < PointN
```

(c.f. [11, 33]). This leads to the following subtyping rule for object types:

$$\frac{\Gamma \vdash N' \leq N \qquad \Gamma \vdash \mathit{Object}(N') \in \star \qquad \Gamma \vdash \mathit{Object}(N) \in \star}{\Gamma \vdash \mathit{Object}(N') \leq \mathit{Object}(N)}$$

The monotonicity of the *Object* constructor captures the intuition that whenever the interface N' of an object type $\mathit{Object}(N')$ is more refined than the interface N of an object type $\mathit{Object}(N)$, elements of $\mathit{Object}(N')$ should be allowed in contexts where elements of $\mathit{Object}(N)$ are expected.

Next, we consider generic methods. Observe that if we simply apply the generic `setX` of points to an element of `CPoint` (which is valid by the rule of subsumption), the result will be an element of `Point`, not of `CPoint`. In the presence of subtyping, our generic methods are insufficiently polymorphic. More generally, suppose that N is a functorial operator and $N' \leq N$.[3] The application of GM_N to an element of $\mathit{Object}(N')$ should yield an element of $N(Object(N'))$, not $N(Object(N))$ as above. This suggests a change in the type of GM (c.f. [14]):

$$\frac{\Gamma \vdash N \in \star\!\rightarrow\!\star \qquad co(N)}{\Gamma \vdash GM_N \in \mathit{All}(N'{\leq}N)\ \mathit{Object}(N') \rightarrow N(\mathit{Object}(N'))}$$

The interaction between the subtype relation and the term constructors *object* and *GM* is axiomatized by a rule like Eq-Str-Nat, which stipulates that they should commute, when all the operators involved are functorial.[4] (More generally, we might require that every well-typed equation whose type-erasure is a syntactic identity should be provable in the equational theory; c.f. [13, 27].)

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \qquad \Gamma \vdash \mathit{Object}(N') \in \star \\ co(N') \qquad co(N) \\ \Gamma \vdash R' \leq R \qquad \Gamma \vdash s \in R' \\ \Gamma \vdash m \in R \rightarrow N'(R')\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{object}_{N'}(R', s, m) \\ \quad = \mathit{object}_N(R, s, m) \\ \quad \in \mathit{Object}(N)\end{array}} \qquad \text{(Eq-Object-Nat)}$$

[3] Note that this does not imply that N' is also functorial.

[4] The problem of constraining the behavior of $GM_N[N']$ for arbitrary N' is a matter for further research. It seems to be related to the issues discussed in [29].

$$\frac{\Gamma \vdash N' \leq N \in \star \!\rightarrow\! \star \qquad co(N') \qquad co(N)}{\Gamma \vdash GM_{N'} = GM_N \in \mathit{All}(N'' \!\leq\! N')\ \mathit{Object}(N'') \rightarrow N(\mathit{Object}(N''))} \qquad (\textsc{Eq-GM-Nat})$$

3.1. Example: These conditions imply an equation similar to the one given in Mitchell's treatment of method specialization and inheritance via natural transformations [26]:

$$\begin{tikzcd}
\mathit{Object}(N) \arrow[r, "GM_N[N]"] & N(\mathit{Object}(N)) \\
\mathit{Object}(N') \arrow[u, "\leq"] \arrow[r, "GM_{N'}[N']"'] & N'(\mathit{Object}(N')) \arrow[u, "\leq"']
\end{tikzcd}$$

Combining this diagram with the specification (†), we obtain the following generalized characterization of the behavior of the generic method, capturing the idea that it does nothing but invoke the concrete methods of the object to which it is applied:

$$\begin{tikzcd}
\mathit{Object}(N') \arrow[r, "GM_N[N']"] & N(\mathit{Object}(N')) \\
 & N'(\mathit{Object}(N')) \arrow[u, "\leq"'] \\
R \arrow[uu, "\mathit{object}_{N'}(R, \text{—}, m)"] \arrow[r, "m"'] & N'(R) \arrow[u, "\substack{\mathit{strength}_{N'}\,[R]\,[\mathit{Object}(N')] \\ \mathit{object}_{N'}(R, \text{—}, m)}"']
\end{tikzcd} \qquad (\dagger_{\leq})$$

4 High-Level Syntax

This framework yields a natural, uniform syntax for declarations of object types. For example, the declaration

```
Point =
  ObjectType(Rep) with
    setX: Int->Rep,
    getX: Int
```

stands for the following set of declarations:

```
PointN  =  Fun(Rep) {setX:Int->Rep, getX:Int}
Point'setX = fun(N<PointN) fun(p:Object(N)) (GM_PointN[N](p)).setX
Point'getX = fun(N<PointN) fun(p:Object(N)) (GM_PointN[N](p)).getX
```

Similarly,

```
CPoint =
  ObjectType(Rep) with
    setX: Int->Rep,
    getX: Int,
    setC: Color->Rep,
    getC: Color
```

stands for

```
CPointN  =  Fun(Rep) {setX:Int->Rep,
                      getX:Int,
                      setC:Color->Rep,
                      getC:Color}
CPoint'setX = fun(N<CPointN) fun(p:Object(N)) (GM_CPointN[N](p)).setX
CPoint'getX = fun(N<CPointN) fun(p:Object(N)) (GM_CPointN[N](p)).getX
CPoint'setC = fun(N<CPointN) fun(p:Object(N)) (GM_CPointN[N](p)).setC
CPoint'getC = fun(N<CPointN) fun(p:Object(N)) (GM_CPointN[N](p)).getC
```

This translation has been implemented in a prototype typechecker based on the encoding of objects with existential types given the next section [33].

5 Objects as Packages

We now consider a specific encoding of objects, where existential types are used to achieve the hiding of the internal states of objects [20, 25, 33]:[5]

$$Object(N) \stackrel{\text{def}}{=} Some(A)\ \{state : A,\ methods : A \rightarrow N(A)\}$$

or, in the more familiar notation:

$$Object(N) = \exists A.\ A \times (A \rightarrow N(A)).$$

The rules for existential types in $F^{\omega}_{\leq}$ are summarized in Appendix A.8, following [14] and [28]. The definition of *strength* for this calculus, a straightforward extension of the definition of *strength* for pure $F^{\omega}_{\leq}$, is given in Appendix A.7.

5.1. Definition: The type-theoretic encoding of objects in $F^{\omega}_{\leq}$ using existentials is given by:

$$Object(N) \stackrel{\text{def}}{=} Some(A)\ \{state : A,\ methods : A \rightarrow N(A)\}$$

$$\begin{aligned} object_N(R,\ s,\ m) \stackrel{\text{def}}{=}\ & pack\ \{state = s,\ methods = m\} \\ & as\ Object(N) \\ & hiding\ R \end{aligned}$$

$$\begin{aligned} GM_N \stackrel{\text{def}}{=}\ & fun(N' \leq N) \\ & \quad fun(x : Object(N')) \\ & \qquad open\ x\ as\ [R, r]\ in \\ & \qquad\quad (strength_N\ [R]\ [Object(N')] \\ & \qquad\qquad object_{N'}(R,\ \text{—},\ r.methods)) \\ & \qquad\quad (r.methods\ r.state) \end{aligned}$$

[5] Also c.f. [6], which is mainly based on recursive records, but where existential types are used to implement hidden instance variables.

Note that the encoding of object types and the packing function *object* makes sense for arbitrary N — indeed, even for the non-unary interface types we excluded in Section 2 — but the definition of the generic method relies on the functoriality of N.

5.2. Lemma: This encoding satisfies the object axioms in Appendix B.

Proof: Diagram (†) follows by the rules EQ-SOME-BETA and EQ-RCD-SURJ from Appendix A. Rule EQ-OBJECT-NAT follows by EQ-PACK and the existential analog of EQ-ABS'. Rule EQ-GM-NAT follows by EQ-OPEN and EQ-STR-NAT. □

This encoding is interesting both because it works in a fairly simple calculus — pure $F^\omega_\leq$ enriched with just existential types (which, in fact, can be encoded in pure $F^\omega_\leq$) — and because it avoids introducing the possibility of nontermination in situations where fixed points are not strictly required.

6 Objects as Recursive Records

Next we show that the familiar encoding of objects as recursive records [9, 12, 6, 26, etc.] also satisfies the abstract specification developed in Sections 2 and 3. This justifies both the abstract framework itself (by showing that a familiar construction is a specific instance of it) and the encoding (by showing that some of its tricky aspects, e.g. the creation of objects, can be explained from general considerations).

We extend the pure $F^\omega_\leq$ of Sections A.1 through A.5 with a recursive type constructor μ, obeying the subtyping laws:

$$\frac{\Gamma \vdash \mu(A)T \in \star}{\Gamma \vdash \mu(A)T \lesseqgtr [(\mu(A)T)/A]T} \quad \text{(S-FOLD)}$$

$$\frac{\Gamma,\, B\leq Top(\star),\, A\leq B \vdash S \leq T}{\Gamma \vdash \mu(A)S \leq \mu(B)T} \quad \text{(S-MU)}$$

This extension is summarized in Appendix A.9; for more details, see [2].

We also assume the existence of a fixed-point combinator:

$$\mathit{fix} \in \mathit{All}(A\leq Top(\star))\ (A\to A) \to A$$

This does not add any computational power, since it can be defined using contravariant μ-types [2].

The definitions of positivity and *strength* for the extended system appear in Appendix A.9. Somewhat surprisingly, a type variable A can be said to appear positively in a type $T = \mu(B)F$ only if *both* A and B appear only positively in F.

The type constructor *Object* can be encoded in this calculus by taking

$$\mathit{Object}(N) \stackrel{\text{def}}{=} \mu(X)\ N(X),$$

reflecting the intuition that the extension of an object comprises the potential results of all methods applicable to it. (This is analogous to the observation, captured formally by the rule EQ-ETA, that the extension of a function is its input-output behavior.)

The *object* constructor is implemented using the equational specification (†) as a recursive definition:

$$\mathit{object}_N(R,\, s,\, m) \stackrel{\text{def}}{=} \mathit{obj}\ s$$

where

$$
\begin{aligned}
obj \quad = \quad & \mathit{fix}\,[R{\rightarrow}\mathit{Object}(N)] \\
& \quad \mathit{fun}\,(f{:}R{\rightarrow}\mathit{Object}(N)\,) \\
& \quad\quad \mathit{fun}\,(s{:}R) \\
& \quad\quad\quad (\mathit{fold} \ \in \ N(\mathit{Object}(N)){\rightarrow}\mathit{Object}(N)) \\
& \quad\quad\quad\quad (\mathit{strength}_N\ [R]\ [\mathit{Object}(N)]\ f\ (m\ s))
\end{aligned}
$$

(The function *fold* is actually an implicit coercion; we write it explicitly here for extra clarity.)

Finally, since $\mathit{Object}(N) \lessgtr N(\mathit{Object}(N))$, (by S-MU) the generic method is just an identity function:

$$
\begin{aligned}
GM_N \ \stackrel{\text{def}}{=} \ & \mathit{fun}(N'{\le}N)\ \ \mathit{id}\,[\mathit{Object}(N')] \\
\in \ & \mathit{All}(N'{\le}N)\ \ \mathit{Object}(N') \ \rightarrow\ \mathit{Object}(N') \\
\lessgtr \ & \mathit{All}(N'{\le}N)\ \ \mathit{Object}(N') \ \rightarrow\ N'(\mathit{Object}(N')) \\
\le \ & \mathit{All}(N'{\le}N)\ \ \mathit{Object}(N') \ \rightarrow\ N(\mathit{Object}(N')).
\end{aligned}
$$

For example, consider the signature of points as defined above:

```
PointN(X) = {getX:Int, setX:Int->X}.
```

Suppose we are given the representation type `Int` and the following implementation of the point methods:

```
m = fun(r:Int)
      {getX = r,
       setX = fun(i:Int) i}
   : Int -> PointN(Int)
```

Then by unravelling the definition of *object*, we obtain a function `mkpoint` mapping internal states to point objects as follows:

```
mkpoint = fix [Int->Point]
            fun(mkp: Int->Point)
              fun(s:Int)
                (fold
                  {getX = s,
                   setX = fun(i:Int) mkp i})
```

6.1. Remark: The definition $\mathit{Object}(N) \stackrel{\text{def}}{=} \mu(X)\ N(X)$ and the use of identity functions for the generic method also make sense for mixed-variance N. However, it is not possible to define the packing function *object* uniformly in this case. Conversely, we have already observed that existentials allow the uniform definition of the *object* function for arbitrary M but possess generic methods only in the covariant case. Thus, the encodings using existentials and recursive types offer complementary ways of extending our account to mixed-variance method signatures. We conjecture that the arguments in [3] and in Freyd's email notes on "Structors" can be generalized to show that, in the Bruce/Mitchell model [7], the encodings using recursive types and existential types are isomorphic for covariant N.

7 F-bounded Quantification

If we model objects using recursive types, then we can eliminate the higher-order quantification in the type of the generic method in favor of a specialized form of second-order quantification called *F-bounded quantification* [8, 17], where the type variable introduced by a quantifier may appear free in its bound.

Cardelli and Mitchell have observed that F-bounded quantification can be expressed in terms of higher-order quantification and recursive types [1, 6]:

$$All(A \leq F(A))\ S \quad \approx \quad All(G \leq F)\ \ [(\mu(A)G(A))/A]S.$$

Indeed, it follows from an observation by Abadi [1] that, in many models, these two types denote the same collection.

Using this correspondence, we can recast the type of our generic method as:

$$GM_N \in All(A \leq N(A))\ A \rightarrow N(A)$$

This typing also appears in [17].

8 Parametricity

Under the assumption that all definable functions of polymorphic type satisfy a suitable parametricity criterion,[6] we can give a more intuitive justification of the equations in Section 2. We drop subtyping here and in the following section, so that our development can be carried out in a better-understood formal framework.)

The idea is to look at observable behavior. Suppose we are given a function

$$c \in Object(N) \rightarrow Obs,$$

where *Obs* is some observable type, e.g. `Bool`.[7] Now, if objects are properly abstract, then c can only inspect its argument by applying the generic method GM_N. In this case, it should be possible to replace GM_N in the body of c by any other function of the same shape. More precisely:

8.1. Observation: Every $c \in Object(N) \rightarrow Obs$ has the form

$$c = c'\ [Object(N)]\ GM_N$$

where c' is a parametric [37, 38] polymorphic function of type

$$c' \in All(X)\ \ (X \rightarrow N(X)) \rightarrow X \rightarrow Obs.$$

For example, the function

```
testPoint = fun(p:Point) eqInt (GM_Point (GM_Point p).setX 7).getX 7
```

can be rewritten as

[6] It is known [37] that all polymorphic functions definable in System F are parametric in the required sense; we expect that this result can be extended to the calculi we use here (c.f. [13]).

[7] A more general definition of "observable behavior" would consider functions of type $(A \rightarrow Object(N)) \rightarrow Obs$ for an arbitrary type A representing possible free variables. We believe that the following development can also be carried out in this generalized context.

```
testPoint = testPoint' [Point] GM_Point,
```

where

```
testPoint' = fun(P) fun(gp:P->PointN(P))
               fun(p:P)
                 eqInt (gp ((gp p).setX 7)).getX 7.
```

This leads to an alternative specification of objects.

8.2. Definition: A type system providing *Object*, *object*, and *GM* as in Section 2 is said to satisfy condition PARA if, given a functorial operator N and

$$\begin{array}{lcl} c' & \in & All(X)\ (X \rightarrow N(X)) \rightarrow X \rightarrow Obs \\ R & \in & \star \\ m & \in & R \rightarrow N(R) \\ s & \in & R, \end{array}$$

we have

$$c'\ [Object(N)]\ GM_N\ object_N(R, s, m) \ = \ c'\ [R]\ m\ s. \qquad \text{(PARA)}$$

In view of the discussion above, this says that objects and their representations are interchangeable in observable contexts.

8.3. Proposition: If an implementation of objects meets the equational specification (†), then it also satisfies PARA.

Proof: Let N, c, R, s, and m be as above. The assumption that c' is parametric implies that, for any two types X and Y, if

$$\begin{array}{l} f \in X \rightarrow N(X) \\ g \in Y \rightarrow N(Y) \\ h \in X \rightarrow Y \end{array}$$

satisfy

$$f\ ;\ strength_N[X][Y](h) = h\ ;\ g$$

then for every $x \in X$,

$$c'\ [X]\ f\ x \ = c'\ [Y]\ g\ (h\ x).$$

(This follows from Reynolds' definition of parametricity [37] by considering the graph of h as a relation. This simplified version of parametricity avoids the need for a general definition of "relation" in an arbitrary type theory.) Now let

$$\begin{array}{lcl} X & = & R \\ Y & = & Object(N) \\ f & = & m \\ g & = & GM_N \\ h & = & object_N(R, —, m). \end{array}$$

Then (†) yields

$$f\ ;\ strength_N[X][Y](h) = h\ ;\ g$$

as required. $\square$

The converse also holds, under the assumption that equality is given by observational equivalence.

8.4. Definition: Two terms $s, t \in A$ are said to be observationally equivalent, written $s =_{\text{obs}} t$, if, for all functions $c \in A \rightarrow Obs$, we have $c\ s = c\ t$.

8.5. Proposition: If an implementation of objects meets the parametric specification PARA and equality on objects is given by observational equivalence, then for any functorial N and for

$$\begin{array}{lll} R & \in & \star \\ m & \in & R \rightarrow N(R) \\ s & \in & R, \end{array}$$

the following equation holds:

$$\begin{array}{ll} & GM_N\ object_N(R,\ s,\ m) \\ =_{\text{obs}} & strength_N\ [R]\ [Object(N)]\ object_N(R,\ —,\ m)\ (m\ s) \end{array}$$

For the proof, we need the following lemma, establishing an η-like equation for objects up to observational equivalence.

8.6. Lemma: Under condition PARA, for all $c \in Object(N) \rightarrow Obs$ and $x \in Object(N)$, we have

$$c\ x\ =\ c\ object_N(Object(N),\ x,\ GM_N).$$

Proof: By Observation 8.1, there is some $c' \in All(X)\ (X \rightarrow N(X)) \rightarrow X \rightarrow Obs$ such that the right-hand side can can be written as

$$c'\ [Object(N)]\ GM_N\ object_N(Object(N),\ x,\ GM_N).$$

By PARA, this is equal to $c'\ [Object(N)]\ (GM_N\ x)$, which equals the left-hand side. $\square$

Proof of Proposition 8.5: Now choose any $c \in N(Object(N)) \rightarrow Obs$. We must show:

$$\begin{array}{ll} & c\ (GM_N\ object_N(R,\ s,\ m)) \\ = & c\ (strength_N\ [R]\ [Object(N)]\ object_N(R,\ —,\ m)\ (m\ s)). \end{array}$$

Let

$$\begin{array}{lll} c' & \in & All(X)\ (X \rightarrow N(X))\ \rightarrow\ X\ \rightarrow\ Obs \\ & \stackrel{\text{def}}{=} & fun(X)\ fun(m{:}X \rightarrow N(X))\ fun(x{:}X) \\ & & \quad c\ (strength_N\ [X]\ [Object(N)]\ object_N(X,\ —,\ m)\ (m\ x)). \end{array}$$

Then the right-hand side above is $c'\ [R]\ m\ s$. By PARA, this equals

$$c'\ [Object(N)]\ GM_N\ object_N(R,\ s,\ m),$$

which expands to

$$\begin{array}{l} c\ (strength_N\ [Object(N)]\ [Object(N)] \\ \quad object_N(Object(N),\ —,\ GM_N) \\ \quad (GM_N\ object_N(R,\ s,\ m))). \end{array}$$

By Lemma 8.6 and the assumption that equality is given by observational equivalence, the function in the argument to *strength* is the identity. So by the functoriality of N, this expression equals the left-hand side. □

8.7. Remark: Although we could have specified the generic method directly using this condition, we do not know how to generalize the argument to include subtyping. We therefore prefer to take the specification (†) as primary and regard the characterization in terms of observable behavior as a justification of its naturalness.

9 Mixed-Variance Signatures and Packages

This completes the main thread of our discussion. In the next two sections, we consider extending our theory to include mixed-variance object interfaces. Although we are doubtful as to the practical importance of this more general case, the discussion sheds some additional light on the formal relation between the object encoding in terms of packages and the one using recursive records.

Suppose we are given a mixed-variance method signature, say

```
EqPointN(X) = {setX:Int->X, getX:Int, eq:X->bool}.
```

Then, as we have seen, we cannot expect to have both the generic method and the packing function *object* in their full generality. We have also seen that if we suitably weaken either requirement we can obtain a satisfactory account. In this section we show how to handle this problem more abstractly, by deriving a solution in terms of purely covariant signatures.

We again state our requirements first in an abstract form and try to give a suitable axiomatisation. Let $N \in \star{\rightarrow}\star$ be a mixed-variance type operator. It is known that N can always be written in the form $N = Fun(X)\ \ F\ X\ X$ for some binary operator F that is contravariant in its first and covariant in its second argument. For example, for the signature of points with equality we obtain

```
EqPointF = Fun(X,Y) {setX:Int->Y, getX:Int, eq:X->bool}.
```

9.1. Definition: We write $mixed(F)$ to indicate that is F contravariant in its first argument and covariant in its second argument:

$$mixed(F) \text{ iff } contra(Fun(X)\ \ FXY) \text{ and } co(Fun(Y)\ \ FXY).$$

The well-formedness rule for object types and the typing of the generic method are straightforward generalizations of the ones in Section 2:

$$\frac{\Gamma \vdash F \in \star{\rightarrow}\star{\rightarrow}\star}{CObject(F) \in \star}$$

$$\frac{\Gamma \vdash F \in \star{\rightarrow}\star{\rightarrow}\star \qquad mixed(F)}{\Gamma \vdash CGM_F \in CObject(F) \rightarrow (F\ CObject(F)\ CObject(F))}$$

For example:

```
CGM_EqPointF : EqPoint -> {setX: EqPoint->Int,
                           getX: Int,
                           eq: EqPoint->Bool},
```

where

```
EqPoint = CObject(EqPointF).
```

In the introduction rule, it is not enough to implement the methods only with respect to the representation type: a concrete equality method must be able to cope with a second argument whose implementation uses on a different representation type. Since, in a sense, $CObject(F)$ subsumes all other representation types, we might consider an introduction rule of the form:

$$\frac{\begin{array}{c}\Gamma \vdash F \in \star\!\to\star\to\star \qquad mixed(F) \\ \Gamma \vdash R \in \star \qquad \Gamma \vdash s \in R \qquad \Gamma \vdash m_o \in R \to (F\ CObject(F)\ R)\end{array}}{\Gamma \vdash cobject_F(R, s, m_o) \in CObject(F)}$$

But an m_o in the premise of this rule can only access objects via the generic method CGM_F. So, as we observed in Section 8, there must be some

$$m \in R \to (All(Y)\ (Y \to FYY) \to FYS)$$

such that $m_o = m\ [CObject(F)]\ CGM_F$. By using this underlying polymorphic function directly, we arrive at the following refined introduction rule:

$$\frac{\begin{array}{c}\Gamma \vdash F \in \star\!\to\star\to\star \qquad mixed(F) \\ \Gamma \vdash R \in \star \qquad \Gamma \vdash s \in R \\ \Gamma \vdash m \in R \to (All(Y)\ (Y \to FYY) \to FYR)\end{array}}{\Gamma \vdash cobject_F(R, s, m) \in CObject(F)}$$

For example, point objects with equality can be created by

```
mkeqpoint = fun(s:Int) cobject_EqPointF(Int, s, m),
```

where

```
m = fun(s:Int)
      fun(Y) fun(ym: Y->EqPointF(Y,Y))
        {setX = fun(i:Int) i,
         getX = s,
         eq = fun(y:Y) eqInt (ym y).getX s}.
```

The observation that methods can only manipulate other objects generically also underlies the equational specification of GM. If R is some representation type and $m \in R \to All(Y)\ (Y \to FYY) \to FYS$ is an implementation of the methods according to the above rule, then the specification (†) generalizes to:

$$\begin{tikzcd}[column sep=huge, row sep=huge] O \arrow[r, "CGM_F"] & FOO \\ R \arrow[u, "{cobject_F(R,\,—,\,m)}"] \arrow[r, "{m\ —\ [O]\ CGM_F}"'] & FOR \arrow[u, "{strength_{(FO-)}\ [R]\ [CObject(F)]\ cobject_F(R,\,—,\,m)}"'] \end{tikzcd} \tag{‡}$$

where $O = CObject(F)$.

In the case of point objects with equality, this expands to the statement that the generic equality method applied to two points should yield the same result as the application of the local equality method of the first object to the second object:

```
    (CGM_EqPointF (mkpoint x)).eq y
=
    eqInt (CGM_EqPointF y).getX x.
```

This again reflects the intuition that the generic method should not perform any computations, but should simply invoke the local implementation.

It may surprise the reader to observe that an implementation of mixed-variance objects satisfying this specification can be used to solve fixed-point equations. Let $\phi \in D \rightarrow D$ be a function whose fixed point we wish to calculate, and let

$$
\begin{array}{lcl}
F\ X\ Y & = & X \rightarrow D \\
O & = & \mathit{CObject}(F) \\
g & = & \mathit{CGM}_F \\
 & \in & \mathit{CObject}(F) \rightarrow \mathit{CObject}(F) \rightarrow D.
\end{array}
$$

We can build an element of O using the type Unit (or any other inhabited type) as representation type and the function

$$
\begin{array}{lcl}
m & = & \mathit{fun}(x{:}\mathit{Unit})\ \mathit{fun}(Y)\ \mathit{fun}(h{:}Y \rightarrow Y \rightarrow D)\ \mathit{fun}(y{:}Y)\ \phi(h\ y\ y) \\
 & \in & \mathit{Unit} \rightarrow (\mathit{All}(Y)\ (Y \rightarrow Y \rightarrow D) \rightarrow Y \rightarrow D)
\end{array}
$$

as the concrete method:

$$
\begin{array}{lcl}
o & = & \mathit{cobject}_F(\mathit{Unit},\ \mathit{unit},\ m) \\
 & \in & O,
\end{array}
$$

where unit is the unique element of Unit. It follows from (‡) that $(g\ o\ o)$ is a fixed point of ϕ. To see this, observe that F is constant in its positive argument, so $\mathit{strength}_{(FO-)}\ [U]\ [V]\ f$ is the identity function on $O \rightarrow D$ no matter what $f \in U \rightarrow V$ is. So we can calculate as follows:

$$
\begin{array}{ll}
 & g\ o\ o \\
= & g\ (\mathit{cobject}_F(\mathit{Unit},\ \mathit{unit},\ m))\ o \\
= & \mathit{id}_{(O \rightarrow D)}\ (m\ \mathit{unit}\ [O]\ g\ o) \\
= & \phi\ (g\ o\ o).
\end{array}
$$

Thus, the above typing rules imply the existence of a fixed-point combinator on the level of values. Conversely, given such a fixed-point combinator and an implementation of objects for purely covariant signatures, we can implement the above specified mixed-variance objects as follows:

$$
\begin{array}{lcl}
F^{\star} & \stackrel{\mathrm{def}}{=} & \mathit{Fun}(X)\ \mathit{All}(Y)\ (Y \rightarrow FYY) \rightarrow FYX \\
\mathit{CObject}(F) & \stackrel{\mathrm{def}}{=} & \mathit{Object}(F^{\star}) \\
\mathit{CGM}_F & \stackrel{\mathrm{def}}{=} & \mathit{fix}(\mathit{cgm} : \mathit{Object}(F^{\star}) \rightarrow (F\ \mathit{Object}(F^{\star})\ \mathit{Object}(F^{\star}))) \\
 & & \quad \mathit{fun}(\mathit{ob}{:}\mathit{Object}(F^{\star})) \\
 & & \qquad\quad \mathit{GM}_{F^{\star}}\ \mathit{ob}\ [\mathit{Object}(F^{\star})]\ \mathit{cgm} \\
\mathit{cobject}_F(R,\ s,\ m) & \stackrel{\mathrm{def}}{=} & \mathit{object}_{F^{\star}}(R,\ s,\ m)
\end{array}
$$

It follows by simple calculation that these definitions meet our above specifications.

Now if we use the encoding of covariant objects as existential types, we arrive at an account of binary methods that does without recursive types. For example, the covariant interface for points with equality is:

```
EqPointN = Fun(B)
              All(A)
                (A -> {setX:Int->A, getX:Int, eq:A->Bool}) ->
                {setX:Int->B, getX:Int, eq:A->Bool}
```

The encoding of objects with this signature in terms of existential types is:

```
EqPoint
  = Some(B)
       {state: B,
        methods: B -> All(A)
                        (A -> {setX:Int->A, getX:Int, eq:A->Bool}) ->
                        {setX:Int->B, getX:Int, eq:A->Bool}}
```

The creation function is:

```
object_EqPointN
  = fun(R)
      fun(s : R)
        fun(m : R -> All(A)
                       (A -> {setX:Int->A, getX:Int, eq:A->Bool}) ->
                       {setX:Int->R, getX:Int, eq:A->Bool})
          pack
            {state = s,
             methods = m}
          as EqPoint
          hiding R
```

New points are created by supplying implementations as before, where the concrete methods are suitably abstracted:

```
p1 = object_EqPointN [Int] 5
       fun(s : Int)
       fun(A)
       fun(n : A -> {setX:Int->A, getX:Int, eq:A->Bool})
         {setX = fun(i:Int) i,
          getX = s,
          eq   = fun(q:A) eqInt s ((n q).getX)}
```

The generic method is:

```
CGM_EqPointF
  = rec [EqPoint -> {setX:Int->EqPoint, getX:Int, eq:EqPoint->Bool}]
     fun(cgm : EqPoint -> {setX:Int->EqPoint, getX:Int, eq:EqPoint->Bool})
      fun(ob : EqPoint)
       open ob as [R,r] in
          (strength_EqPointN [R] [EqPoint]
             (fun(s:R) object_EqPointN [R] (s) (r.methods)))
          [EqPoint]
          (r.methods r.state [EqPoint])
          cgm
```

Now, if we name the components of the generic method as three separate functions,

```
EqPoint'setX = fun(p:EqPoint) (CGM_EqPointF p).setX
EqPoint'getX = fun(p:EqPoint) (CGM_EqPointF p).getX
EqPoint'eq =   fun(p:EqPoint) (CGM_EqPointF p).eq
```

we can calculate as follows:

```
EqPoint'eq p1 (EqPoint'setX p1 3)
  = false

EqPoint'eq p1 (EqPoint'setX p1 5)
  = true.
```

Unfortunately, we encounter difficulties when we try to extend this account to include subtyping, as we did in Section 3. There does not appear to be any natural way to implement a generic method abstracted over the subtypes of a mixed-variance specification — either abstractly in terms of the covariant object facilities, or even directly in terms of existential types — since the implementation of the mixed-variance generic method relies on the functorial strength of the sub-specification, which is not available for arbitrary operators.

We can obtain a rather unnatural generic method by abstracting explicitly on the required strength as well as the sub-specification:

$$\begin{array}{lll} CGM_F & \in & All(F' \leq F) \\ & & \quad (All(X,Y,Z)\ (Y \rightarrow Z) \rightarrow (F\,X\,Y \rightarrow F\,X\,Z)) \rightarrow \\ & & \qquad CObject(F') \rightarrow (F\ CObject(F')\ CObject(F')) \end{array}$$

This again suggests the naturalness of the refined type theory we mentioned in the footnote on p. 5.

10 Mixed-Variance Signatures and Recursive Records

Of course, recursive records can also be used to implement the underlying covariant object facilities, leading to a similar account of mixed-variance objects in terms of recursive types. But this account is not the most straightforward one possible, and does not give rise to the familiar examples from the literature. In this section we present a more direct extension of our framework to mixed-variance signatures, from which the familiar recursive records follow immediately. This extension also handles subtyping.

We begin by returning to the somewhat circular introduction rule for object types

$$\frac{\begin{array}{c} \Gamma \vdash F \in \star \rightarrow \star \rightarrow \star \qquad mixed(F) \\ \Gamma \vdash S \in \star \qquad \Gamma \vdash s \in S \qquad \Gamma \vdash m \in S \rightarrow (F\ CObject(F)\ S) \end{array}}{\Gamma \vdash cobject_F(S,\, s,\, m) \in CObject(F)}$$

which we dropped in Section 9 in favor of the rule involving polymorphic methods. The typing of the generic method is now extended to handle subtyping exactly as in Section 3:

$$\frac{\Gamma \vdash F \in \star \rightarrow \star \rightarrow \star \qquad mixed(F)}{\Gamma \vdash CGM_F \in All(F' \leq F)\ CObject(F') \rightarrow (F\ CObject(F')\ CObject(F'))}$$

For example, the generic method for points with equality now has the type:

```
CGM_EqPointF : All(F'<EqPointF)
                 CObject(F') -> {setX: CObject(F')->Int,
                                 getX: Int,
                                 eq: CObject(F')->Bool},
```

Elements of `CObject(EqPointF)` can be created using a `mkeqpoint` method similar to the one in the previous section:

```
mkeqpoint = fun(s:Int) cobject_EqPointF(Int, s, m),
```

where

```
m = fun(s:Int)
        {setX = fun(i:Int) i,
         getX = s,
         eq = fun(y: CObject(EqPointF))
                eqInt (CGM_EqPointF [EqPointF] y).getX s}.
```

Finally, subtyping between object types must be defined by a more restrictive rule than the one we used in Section 3:

$$\frac{\Gamma,\, B{\leq}\mathit{Top}(\star),\, A{\leq}B \;\vdash\; F'AA \;\leq\; FBB}{\Gamma \;\vdash\; \mathit{CObject}(F') \;\leq\; \mathit{CObject}(F)}$$

That is, we require F' to be *monotonically* a subtype of F, not just pointwise. To see why this is necessary, consider

```
EqCPointF = Fun(X,Y) {setX: Int->Y, getX: Int, eq: X->bool,
                      setC: Color->Y, getC: Color}.
```

If we allowed `CObject(EqCPointF)` $\leq$ `CObject(EqPointF)`, then we could break the type system by creating an instance `cp` of `CObject(EqCPointF)` whose equality method called the `getC` method of its argument, promoting `cp` to type `CObject(EqPointF)`, and invoking its `eq` method with an instance of `CObject(EqPointF)` as argument (c.f. [6]).

Both this subtyping rule and the packing rule above strongly suggest recursive types (c.f. [2]). Indeed, an encoding of objects satisfying this specification can be given very directly in terms of recursive record types:

$$
\begin{array}{lll}
\mathit{CObject}(F) & \stackrel{\text{def}}{=} & \mu(X)\ FXX \\
 & & \\
\mathit{CGM}_N & \stackrel{\text{def}}{=} & \mathit{fun}(F'{\leq}F)\ \ \mathit{id}\,[\mathit{CObject}(F')] \\
 & \in & \mathit{All}(F'{\leq}F)\ \ \mathit{CObject}(F') \to \mathit{CObject}(F') \\
 & \lessgtr & \mathit{All}(F'{\leq}F)\ \ \mathit{CObject}(F') \to (F'\ \mathit{CObject}(F')\ \mathit{CObject}(F')) \\
 & \leq & \mathit{All}(F'{\leq}F)\ \ \mathit{CObject}(F') \to (F\ \mathit{CObject}(F')\ \mathit{CObject}(F')).
\end{array}
$$

(The bounded quantifier in the type of the generic method refers to pointwise subtyping of type operators, as always — not to monotone subtyping.)

As in Section 6, *cobject* can be defined straightforwardly by considering the specification (‡), as a recursive equation.

11 Conclusions

We have presented a direct, high-level axiomatization of objects and their types in a higher-order polymorphic λ-calculus with subtyping. This framework yields a natural high-level syntax for sending messages to objects and allows several previously studied encodings of objects to be presented in a common setting.

The object encodings using existential types and recursive types coincide for covariant method signatures. The addition of term-level fixed points allows the covariant approach to be extended to mixed-variance signatures, but here the direct formulation in terms of recursive types seems better, since it handles subtyping in a more natural way.

But using mixed-variance signatures to allow binary methods is rather unnatural in the first place. For one thing, by implying the presence of fixed points, it introduces the risk of nontermination even in simple situations like equality methods for points, making the task of proving total correctness unnecessarily difficult. Moreover, the binary methods that can be supported in this way are not what are needed in practice, since they can only access the concrete representation of one of their arguments. It seems better to reject the idea of binary *methods* altogether and use ordinary abstract data types to achieve encapsulation in situations where simultaneous access to the concrete state of more than one datum is required. Indeed, this perspective can be fully integrated with object-style programming, as shown in [31].

We have dealt here only with the basic mechanisms of objects and *subtyping* (a relation between specifications of objects) and not with *inheritance* (a mechanism for deriving the implementation of one class of objects by incrementally modifying the implementation of another class; c.f. [17]). We have argued elsewhere [33] that, once the fundamental mechanisms of encapsulation and subtyping are accounted for and their interactions properly handled, inheritance becomes "just a matter of programming." This leads us to conjecture that it would be a straightforward matter to extend the abstract framework developed here to include an implementation of inheritance, using the ideas developed in [15, 34, 17, 6, 12, 26, 33, 32].

Another application of our framework may lie in suggesting appropriate proof rules for the verification of object-oriented programs. Here, unfortunately, existing work on implementations of inheritance does not seem sufficiently abstract to yield useful *high-level* rules for reasoning about programs involving inheritance. Instead, the process we have described here for objects and subtyping — finding a direct axiomatization and showing that existing implementations can be derived as instances of it — must be repeated for inheritance as well.

Acknowledgements

David N. Turner joined us in many discussions and helped formalize the definitions of positivity and strength in Section 5. Marcelo Fiore pointed out a relationship between terminal algebras and recursive types that led to our axiomatization of mixed-variance methods in Sections 9 and 10. Phil Wadler posed the problem of relating the encoding of objects using recursive records and the one using existential types. Martín Abadi, Eugenio Moggi, and Andre Scedrov provided pointers to relevant literature. Terry Stroup gave us helpful suggestions on an earlier draft.

A Summary of $F^\omega_\leq$

This appendix summarizes the syntax and typing rules of the typed λ-calculus $F^\omega_\leq$, an extension of Girard's system F^ω [21] with subtyping. The ideas behind this system are due to Cardelli, particularly to his 1988 paper, "Structural Subtyping and the Notion of Power Type" [10]; the extension of the subtype relation to type operators was developed by Cardelli and Mitchell [11, 26, 7, 1]. The simple version of $F^\omega_\leq$ used here was obtained as an instance of the definition of a "Generalized Subtyping System" [30], a Generalized Type System in the sense of Barendregt [4] extended with Cardelli's powerkinds. Cardelli's formulation [11] has a more powerful treatment of operator subtyping, including both monotonic and antimonotonic subtyping in addition to pointwise subtyping.

A model for a variant of $F^\omega_\leq$ with recursive types has been given by Bruce and Mitchell [7].

A.1 Syntax

A.1.1. Definition: The sets of kinds, types, terms, and contexts are defined by the following abstract grammar:

K	$::=$	$\star$	kind of types
	$\mid$	$K \rightarrow K$	kind of type operators
T	$::=$	A	type variable
	$\mid$	$\mathit{Fun}\,(A{:}K)\,T$	type operator
	$\mid$	$T\ T$	application of a type operator
	$\mid$	$\mathit{Top}(K)$	top type
	$\mid$	$T \rightarrow T$	function type
	$\mid$	$\mathit{All}\,(A{\leq}T)\,T$	universally quantified type
	$\mid$	$\{l_1{:}T_1\ ..\ l_n{:}T_n\}$	record type
e	$::=$	x	variable
	$\mid$	$\mathit{fun}\,(x{:}T)\,e$	abstraction
	$\mid$	$e\ e$	application
	$\mid$	$\mathit{fun}\,(A{\leq}T)\,e$	type abstraction
	$\mid$	$e\ [T]$	type application
	$\mid$	$\{l_1 = e_1\ ..\ l_n = e_n\}$	record construction
	$\mid$	$e\,.\,l$	field selection
Γ	$::=$	$\bullet$	empty context
	$\mid$	$\Gamma,\ x{:}T$	variable binding
	$\mid$	$\Gamma,\ A{\leq}T$	type variable binding with bound

A.1.2. Notation: The typing rules that follow define sets of valid judgements of the following forms:

$\Gamma \vdash e \in T$	term e has type T
$\Gamma \vdash T \in K$	type T has kind K
$\Gamma \vdash T_1 \leq T_2$	T_1 is a subtype of T_2
$\vdash \Gamma$ context	Γ is a well-formed context

A.1.3. Definition: A type T is *closed* with respect to a context Γ if $\mathrm{FTV}(T) \subseteq \mathrm{dom}(\Gamma)$. A term e is closed with respect to Γ if $\mathrm{FTV}(e) \cup \mathrm{FV}(e) \subseteq \mathrm{dom}(\Gamma)$. A context Γ is closed if

1. $\Gamma \equiv \{\}$, or
2. $\Gamma \equiv \Gamma_1, A{\leq}T$, with Γ_1 closed and T closed with respect to Γ_1, or
3. $\Gamma \equiv \Gamma_1, x{:}T$, with Γ_1 closed and T closed with respect to Γ_1.

A subtyping statement $\Gamma \vdash S \leq T$ is closed if Γ is closed and S and T are closed with respect to Γ; a typing statement $\Gamma \vdash e \in T$ is closed if Γ is closed and e and T are closed with respect to Γ

A.1.4. Convention: In the following, we assume that all statements under discussion are closed. In particular, we allow only closed statements in instances of inference rules. This convention replaces the usual side-conditions in rules such as T-Some-E on p. 33.

A.2 Contexts

$$\vdash \bullet \text{ context} \qquad \text{(C-Empty)}$$

$$\frac{\Gamma \vdash T \in K}{\vdash \Gamma, A{\leq}T \text{ context}} \qquad \text{(C-TVar)}$$

$$\frac{\Gamma \vdash T \in \star}{\vdash \Gamma, x{:}T \text{ context}} \qquad \text{(C-Var)}$$

A.3 Kinding

$$\frac{\Gamma \vdash \Gamma(A) \in K}{\Gamma \vdash A \in K} \qquad \text{(K-TVar)}$$

$$\frac{\Gamma, A{\leq}\mathit{Top}(K_1) \vdash T_2 \in K_2}{\Gamma \vdash \mathit{Fun}(A{:}K_1)\,T_2 \in K_1{\rightarrow}K_2} \qquad \text{(K-Arrow-I)}$$

$$\frac{\Gamma \vdash S \in K_1{\rightarrow}K_2 \qquad \Gamma \vdash T \in K_1}{\Gamma \vdash S\;T \in K_2} \qquad \text{(K-Arrow-E)}$$

$$\frac{\Gamma \text{ context}}{\Gamma \vdash \mathit{Top}(K) \in K} \qquad \text{(K-Top)}$$

$$\frac{\Gamma \vdash T_1 \in \star \qquad \Gamma \vdash T_2 \in \star}{\Gamma \vdash T_1{\rightarrow}T_2 \in \star} \qquad \text{(K-Arrow)}$$

$$\frac{\Gamma, A{\leq}T_1 \vdash T_2 \in \star}{\Gamma \vdash \mathit{All}(A{\leq}T_1)\,T_2 \in \star} \qquad \text{(K-All)}$$

$$\frac{\vdash \Gamma \text{ context} \qquad \text{for each } i,\ \Gamma \vdash T_i \in \star}{\Gamma \vdash \{l_1{:}T_1 \mathinner{..} l_n{:}T_n\} \in \star} \qquad \text{(K-Rcd)}$$

A.4 Subtyping

$$\frac{\Gamma \vdash U \leq S \qquad \Gamma \vdash T \in K \qquad S =_\beta T}{\Gamma \vdash U \leq T} \quad \text{(S-Conv)}$$

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash A \leq \Gamma(A)} \quad \text{(S-TVar)}$$

$$\frac{\Gamma \vdash T \in K}{\Gamma \vdash T \leq T} \quad \text{(S-Refl)}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U} \quad \text{(S-Trans)}$$

$$\frac{\Gamma \vdash T \in K}{\Gamma \vdash T \leq \mathit{Top}(K)} \quad \text{(S-Top)}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma \vdash S_2 \leq T_2 \qquad \Gamma \vdash S_1 \to S_2 \in \star}{\Gamma \vdash S_1 \to S_2 \leq T_1 \to T_2} \quad \text{(S-Arrow)}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma, A \leq T_1 \vdash S_2 \leq T_2 \qquad \Gamma \vdash \mathit{All}(A \leq S_1)\, S_2 \in \star}{\Gamma \vdash \mathit{All}(A \leq S_1)\, S_2 \leq \mathit{All}(A \leq T_1)\, T_2} \quad \text{(S-All)}$$

$$\frac{\{l_1..l_n\} \subseteq \{k_1..k_m\} \qquad \text{for each } k_i = l_j,\ \Gamma \vdash S_i \leq T_j \qquad \Gamma \vdash \{k_1{:}S_1 \mathrel{..} k_m{:}S_m\} \in \star}{\Gamma \vdash \{k_1{:}S_1 \mathrel{..} k_m{:}S_m\} \leq \{l_1{:}T_1 \mathrel{..} l_n{:}T_n\}} \quad \text{(S-Rcd)}$$

$$\frac{\Gamma, A \leq \mathit{Top}(K) \vdash S \leq T}{\Gamma \vdash \mathit{Fun}(A{:}K)\, S \leq \mathit{Fun}(A{:}K)\, T} \quad \text{(S-Abs)}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash S\ U \in K}{\Gamma \vdash S\ U \leq T\ U} \quad \text{(S-App)}$$

A.5 Typing

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T} \quad \text{(T-Subsumption)}$$

$$\frac{\vdash \Gamma \text{ context}}{\Gamma \vdash x \in \Gamma(x)} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash e \in T_2}{\Gamma \vdash \mathit{fun}(x{:}T_1)\, e \in T_1 \to T_2} \quad \text{(T-Arrow-I)}$$

$$\frac{\Gamma \vdash f \in T_1 \to T_2 \qquad \Gamma \vdash a \in T_1}{\Gamma \vdash f\ a \in T_2} \quad \text{(T-Arrow-E)}$$

$$\frac{\Gamma, A \leq T_1 \vdash e \in T_2}{\Gamma \vdash \mathit{fun}(A \leq T_1)\, e \in \mathit{All}(A \leq T_1)\, T_2} \quad \text{(T-All-I)}$$

$$\frac{\Gamma \vdash f \in \mathit{All}(A \leq T_1)\, T_2 \qquad \Gamma \vdash S \leq T_1}{\Gamma \vdash f\, [S] \in [S/A]T_2} \quad \text{(T-All-E)}$$

$$\frac{\vdash \Gamma \text{ context} \qquad \text{for each } i,\ \Gamma \vdash e_i \in T_i}{\Gamma \vdash \{l_1 = e_1 .. l_n = e_n\} \in \{l_1{:}T_1 .. l_n{:}T_n\}} \qquad \text{(T-RCD-I)}$$

$$\frac{\Gamma \vdash e \in \{l{:}T\}}{\Gamma \vdash e.l \in T} \qquad \text{(T-RCD-E)}$$

A.6 Equational Theory

Developing a satisfactory equational theory for $F^{\omega}_{\leq}$ remains a matter for future research. For the sake of concreteness, we propose the following tentative rules, straightforwardly generalizing the equational theory for pure second-order quantification studied by Cardelli, Martini, Mitchell, and Scedrov [13].

$$\frac{\Gamma \vdash e = e' \in T}{\Gamma \vdash e' = e \in T} \qquad \text{(EQ-SYMM)}$$

$$\frac{\Gamma \vdash e = e' \in T \qquad \Gamma \vdash e' = e'' \in T}{\Gamma \vdash e = e'' \in T} \qquad \text{(EQ-TRANS)}$$

$$\frac{\Gamma \vdash x \in T}{\Gamma \vdash x = x \in T} \qquad \text{(EQ-VAR)}$$

$$\frac{\Gamma \vdash e \in \mathit{Top}(\star) \qquad \Gamma \vdash e' \in \mathit{Top}(\star)}{\Gamma \vdash e = e' \in \mathit{Top}(\star)} \qquad \text{(EQ-TOP)}$$

$$\frac{\Gamma, x{:}S \vdash e = e' \in T}{\Gamma \vdash \mathit{fun}\,(x{:}S)\,e = \mathit{fun}\,(x{:}S)\,e' \in S \to T} \qquad \text{(EQ-ABS)}$$

$$\frac{\Gamma \vdash f = f' \in S \to T \qquad \Gamma \vdash a = a' \in S}{\Gamma \vdash f\ a = f'\ a' \in T} \qquad \text{(EQ-APP)}$$

$$\frac{\Gamma, A{\leq}S \vdash e = e' \in T}{\Gamma \vdash \mathit{fun}\,(A{\leq}S)\,e = \mathit{fun}\,(A{\leq}S)\,e' \in \mathit{All}\,(A{\leq}S)\,T} \qquad \text{(EQ-TABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e = e' \in \mathit{All}\,(A{\leq}U)\,V \\ \Gamma \vdash S \leq U \qquad \Gamma \vdash S' \leq U \\ \Gamma \vdash [S/A]V \leq T \qquad \Gamma \vdash [S'/A]V \leq T\end{array}}{\Gamma \vdash e\,[S] = e'\,[S'] \in T} \qquad \text{(EQ-TAPP)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash e_i = e'_i \in T_i}{\Gamma \vdash \{l_1 = e_1 .. l_n = e_n\} = \{l_1 = e'_1 .. l_n = e'_n\} \in \{l_1{:}T_1 .. l_n{:}T_n\}} \qquad \text{(EQ-RCD)}$$

$$\frac{\Gamma, x{:}S \vdash b = b' \in T \qquad \Gamma \vdash a = a' \in S}{\Gamma \vdash (\mathit{fun}\,(x{:}S)\,b)\ a = [a'/x]b' \in T} \qquad \text{(EQ-BETA)}$$

$$\frac{\Gamma \vdash f = f' \in S \to T}{\Gamma \vdash \mathit{fun}\,(x{:}S)\ f\ x = f' \in S \to T} \qquad \text{(EQ-ETA)}$$

$$\frac{\Gamma, A{\leq}S \vdash b = b' \in T \qquad \Gamma \vdash U \leq S}{\Gamma \vdash (\mathit{fun}\,(A{\leq}S)\,b)\ [U] = [U/A]b' \in [U/A]T} \qquad \text{(EQ-TBETA)}$$

$$\frac{\Gamma \vdash f = f' \in \mathit{All}\,(A{\leq}S)\,T}{\Gamma \vdash \mathit{fun}\,(A{\leq}S)\ f\ [A] = f' \in \mathit{All}\,(A{\leq}S)\,T} \qquad \text{(EQ-TETA)}$$

$$\frac{\Gamma \vdash \{l_1 = e_1 \mathinner{..} l_n = e_n\} \in \{l_1{:}T_1 \mathinner{..} l_n{:}T_n\}}{\Gamma \vdash \{l_1 = e_1 \mathinner{..} l_n = e_n\}\,.\,l_i = e_i \in T_i} \quad \text{(Eq-Proj)}$$

$$\frac{\Gamma \vdash r \in \{l_1{:}T_1 \mathinner{..} l_n{:}T_n\}}{\Gamma \vdash r = \{l_1 = r.\ l_1 \mathinner{..} l_n = r.\ l_n\} \in \{l_1{:}T_1 \mathinner{..} l_n{:}T_n\}} \quad \text{(Eq-Surj)}$$

The rule Eq-TApp is closely related to the semantic notion of parametricity.

A.6.1. Conjecture: The following equality rules are derivable (c.f. [13]):

$$\frac{\Gamma \vdash e \in T}{\Gamma \vdash e = e \in T} \quad \text{(Eq-Refl)}$$

$$\frac{\Gamma \vdash e = e' \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e = e' \in T} \quad \text{(Eq-Subsumption)}$$

$$\frac{\Gamma \vdash S' \leq S \qquad \Gamma \vdash T \leq T' \qquad \Gamma, x{:}S \vdash b = b' \in T}{\Gamma \vdash \mathit{fun}(x{:}S)\, b = \mathit{fun}(x{:}S')\, b' \in S' \to T'} \quad \text{(Eq-Abs')}$$

$$\frac{\Gamma \vdash S' \leq S \qquad \Gamma, A{\leq}S' \vdash T \leq T' \qquad \Gamma, A \leq S \vdash b = b' \in T}{\Gamma \vdash \mathit{fun}(A{\leq}S)\, b = \mathit{fun}(A{\leq}S')\, b' \in \mathit{All}(A{\leq}S')\, T'} \quad \text{(Eq-TAbs')}$$

A.7 Functoriality

A.7.1. Definition: Let T be a type that is closed in some context where a type variable A is defined. Then we write $co_A(T)$ when A occurs only positively in T and $contra_A(T)$ when A occurs only negatively in T. These predicates are defined simultaneously as follows:

$co_A(A)$	=	true
$co_A(T)$ with A not free in T	=	true
$co_A(T_1 \to T_2)$	=	$contra_A(T_1)$ and $co_A(T_2)$
$co_A(\mathit{All}(B{\leq}T_1)\, T_2)$	=	A not free in T_1 and $co_A(T_2)$
$co_A(\{l_1{:}T_1, \ldots, l_n{:}T_n\})$	=	$co_A(T_i)$ for all i
$co_A(T)$ in all other cases	=	false
$contra_A(A)$	=	false
$contra_A(T)$ with A not free in T	=	true
$contra_A(T_1 \to T_2)$	=	$co_A(T_1)$ and $contra_A(T_2)$
$contra_A(\mathit{All}(B{\leq}T_1)\, T_2)$	=	A not free in T_1 and $contra_A(T_2)$
$contra_A(\{l_1{:}T_1, \ldots, l_n{:}T_n\})$	=	$contra_A(T_i)$ for all i
$contra_A(T)$ in all other cases	=	false

When $N = \mathit{Fun}(A)\, P$, we write $co(N)$ for $co_A(P)$ and $contra(N)$ for $contra_A(P)$.

A few technical remarks:

- An occurrence of A in the bound of a quantifier is never considered to be either only positive or only negative: $co_A(\mathit{All}(B{\leq}T_1)\, T_2)$ and $contra_A(\mathit{All}(B{\leq}T_1)\, T_2)$ can only hold when T_1 has *no* free occurrences of A. This restriction is necessary because it is not the case, in this calculus, that applying *strength* to a coercion function yields a coercion function. (To see what goes wrong, try extending the *All* case of the definition of *lift* below to handle the case where A occurs in the bound T_1.) The same observation will apply to the bounds of existential quantifiers.

- An application $N(S)$ can only be marked *co* or *contra* if A does not occur free in N or S. A stronger calculus (c.f. [11]) with positive and negative (monotone and antimonotone) operators seems cleaner in this respect, since it allows positivity/negativity to be ascribed to more applications.
- In order that *strength* be defined whenever possible, we need to normalize N before calculating $strength_N$. For example, $strength_{(Fun(X)S)(T)}$ is undefined when A occurs in S or T, but $strength_{[T/X]S}$ may be defined. This normalization step is justified by the derived rule EQ-STR-NAT below.

A.7.2. Definition: Fix a context Γ, a type variable A, two types X and Y, and a function $f \in X{\to}Y$. Then the *lifting* of f through a type T — written $lift_T^{A\leftarrow f}$ or simply $lift_T$ — is

$$
\begin{array}{lcl}
lift_A & = & f \\
lift_S & = & id\,[S] \\
 & & \text{when } A \text{ is not free in } S \\
lift_{T_1\to T_2} & = & fun\,(g : [X/A](T_1{\to}T_2)\,) \\
 & & \quad fun\,(a : [Y/A]T_1) \\
 & & \qquad lift_{T_2}\ (g\ (lower_{T_1}\ a)) \\
lift_{All(B\leq T_1)T_2} & = & fun\,(g : (All\,(B{\leq}T_1)\ [X/A]T_2)\,) \\
 & & \quad fun\,(B{\leq}T_1) \\
 & & \qquad lift_{T_2}\ (g\ [B]) \\
 & & \text{when } A \text{ is not free in } T_1 \\
lift_{\{l_1:T_1,\ \ldots,\ l_n:T_n\}} & = & fun\,(g : [X/A]\{l_1{:}T_1,\ \ldots,\ l_n{:}T_n\}) \\
 & & \quad \{l_1 = lift_{T_1}(g\,.\,l_1),\ \ldots,\ l_n = lift_{T_n}(g\,.\,l_n)\} \\
lift_T & = & \text{undefined in all other cases}
\end{array}
$$

and the *lowering* of f through T, written $lower_T^{A\leftarrow f}$ or simply $lower_T$ is

$$
\begin{array}{lcl}
lower_A & = & \text{undefined} \\
lower_S & = & id[S] \\
 & & \text{when } A \text{ is not free in } S \\
lower_{T_1\to T_2} & = & fun\,(g : [Y/A](T_1{\to}T_2)\,) \\
 & & \quad fun\,(a : [X/A]T_1) \\
 & & \qquad lower_{T_2}\ (g\ (lift_{T_1}\ a)) \\
lower_{All(B\leq T_1)T_2} & = & fun\,(g : (All\,(B{\leq}T_1)\ [Y/A]T_2)\,) \\
 & & \quad fun\,(B{\leq}T_1) \\
 & & \qquad lower_{T_2}\ (g\ [B]) \\
 & & \text{when } A \text{ is not free in } T_1 \\
lower_{\{l_1:T_1,\ \ldots,\ l_n:T_n\}} & = & fun\,(g : [Y/A]\{l_1{:}T_1,\ \ldots,\ l_n{:}T_n\}) \\
 & & \quad \{l_1 = lower_{T_1}(g\,.\,l_1),\ \ldots,\ l_n = lower_{T_n}(g\,.\,l_n)\} \\
lower_T & = & \text{undefined in all other cases.}
\end{array}
$$

Note that $lift_T$ is defined iff $co_A(T)$, that $lower_T$ is defined iff $contra_A(T)$, and that

$$
\begin{array}{lcl}
lift_T & \in & [X/A]T \to [Y/A]T \\
lower_T & \in & [Y/A]T \to [X/A]T.
\end{array}
$$

A.7.3. Definition: When $co(N)$, let

$$strength_N \quad = \quad fun(X)\ \ fun(Y)\ fun(f{:}X{\rightarrow}Y)\ \ lift^{A\leftarrow f}_{N(A)}.$$

(A similar definition is given in [18].) Note that

$$strength_N \in All(X)\, All(Y)\ \ (X{\rightarrow}Y) \rightarrow (N(X){\rightarrow}N(Y)).$$

A.7.4. Lemma: The following equality rules for *lift* and *lower* are derivable from their definitions:

$$\frac{\Gamma,\, A{\leq}Top(*) \ \vdash\ T \in \star \qquad co_A(T) \qquad \Gamma \ \vdash\ X \in \star}{\Gamma \ \vdash\ lift^{A\leftarrow id[X]}_T = id\ [[X/A]T] \in [X/A]T{\rightarrow}[X/A]T} \qquad \text{(Eq-Lift-Id)}$$

$$\frac{\Gamma,\, A{\leq}Top(*) \ \vdash\ T \in \star \qquad contra_A(T) \qquad \Gamma \ \vdash\ X \in \star}{\Gamma \ \vdash\ lower^{A\leftarrow id[X]}_T = id\ [[X/A]T] \in [X/A]T{\rightarrow}[X/A]T} \qquad \text{(Eq-Lower-Id)}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}Top(*),\, \Gamma' \ \vdash\ T \in \star \qquad A \notin \mathrm{FTV}(\Gamma') \qquad co_A(T)\\ \Gamma \ \vdash\ f \in X{\rightarrow}Y \qquad \Gamma \ \vdash\ g \in Y{\rightarrow}Z\end{array}}{\Gamma \ \vdash\ lift^{A\leftarrow(f;g)}_T = (lift^{A\leftarrow f}_T\ ;\ lift^{A\leftarrow g}_T) \in [X/A]T{\rightarrow}[Z/A]T} \qquad \text{(Eq-Lift-Trans)}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}Top(*),\, \Gamma' \ \vdash\ T \in \star\\ A \notin \mathrm{FTV}(\Gamma') \qquad contra_A(T)\\ \Gamma \ \vdash\ f \in X{\rightarrow}Y \qquad \Gamma \ \vdash\ g \in Y{\rightarrow}Z\end{array}}{\begin{array}{l}\Gamma \vdash\ lower^{A\leftarrow(f;g)}_T = (lower^{A\leftarrow g}_T\ ;\ lower^{A\leftarrow f}_T)\\ \quad \in [Z/A]T{\rightarrow}[X/A]\end{array}} \qquad \text{(Eq-Lower-Trans)}$$

A.7.5. Conjecture: The following rules for *lift* and *lower* in the presence of subtyping are derivable:

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}Top(\star) \ \vdash\ T, T' \in \star \qquad co_A(T) \qquad co_A(T')\\ T =_\beta T' \qquad \Gamma \ \vdash\ f \in X{\rightarrow}Y\end{array}}{\Gamma \ \vdash\ lift^{A\leftarrow f}_{T'} = lift^{A\leftarrow f}_T \in [X/A]T' \rightarrow [Y/A]T} \qquad \text{(Eq-Lift-Conv)}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}Top(\star) \ \vdash\ T, T' \in \star\\ contra_A(T) \qquad contra_A(T')\\ T =_\beta T' \qquad \Gamma \ \vdash\ f \in X{\rightarrow}Y\end{array}}{\begin{array}{c}\Gamma \vdash\ lower^{A\leftarrow f}_{T'} = lower^{A\leftarrow f}_T\\ \in [Y/A]T \rightarrow [X/A]T'\end{array}} \qquad \text{(Eq-Lower-Conv)}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}Top(\star) \ \vdash\ T' \leq T\\ co_A(T') \qquad co_A(T) \qquad \Gamma \ \vdash\ f \in X{\rightarrow}Y\end{array}}{\begin{array}{c}\Gamma \vdash\ lift^{A\leftarrow f}_{T'} = lift^{A\leftarrow f}_T\\ \in [X/A]T' \rightarrow [Y/A]T\end{array}} \qquad \text{(Eq-Lift-Sub)}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}Top(\star) \ \vdash\ T' \leq T\\ contra_A(T') \qquad contra_A(T) \qquad \Gamma \ \vdash\ f \in X{\rightarrow}Y\end{array}}{\Gamma \ \vdash\ lower^{A\leftarrow f}_{T'} = lower^{A\leftarrow f}_T \in [Y/A]T \rightarrow [X/A]T'} \qquad \text{(Eq-Lower-Sub)}$$

(The proofs of Eq-Lift-Sub and Eq-Lower-Sub appear to require a transitivity-elimination lemma analogous to the one proved in [5, 19] for $F_{\leq}$.)

A.7.6. Corollary: The definition of *strength* satisfies the functoriality and subtyping laws given in Sections 2 and 3:

$$\frac{\Gamma \vdash N \in \star\to\star \qquad co(N) \qquad \Gamma \vdash X \in \star}{\begin{array}{l}\Gamma \vdash strength_N\ [X]\ [X]\ (id\ [X]) \\ \quad = id\ [N(X)] \\ \quad \in N(X)\to N(X)\end{array}} \qquad \text{(Eq-Str-Id)}$$

$$\frac{\begin{array}{c}\Gamma \vdash N \in \star\to\star \qquad co(N) \\ \Gamma \vdash f \in X\to Y \qquad \Gamma \vdash g \in Y\to Z\end{array}}{\begin{array}{l}\Gamma \vdash strength_N\ [X]\ [Z]\ (f;g) \\ \quad = (strength_N\ [X]\ [Y]\ f)\ ;\ (strength_N\ [Y]\ [Z]\ g) \\ \quad \in N(X)\to N(Z)\end{array}} \qquad \text{(Eq-Str-Trans)}$$

$$\frac{\Gamma \vdash N' \leq N \qquad co(N') \qquad co(N)}{\begin{array}{l}\Gamma \vdash strength_{N'} = strength_N \\ \quad \in All(X \leq Top(\star))\ All(Y \leq Top(\star)) \\ \qquad\qquad (X\to Y) \to (N'(X)\to N(Y))\end{array}} \qquad \text{(Eq-Str-Nat)}$$

A.8 Existentials

To introduce existential quantification, the sets of F^{ω} types and terms are extended as follows:

$$\begin{array}{lcll} T & ::= & \ldots & \\ & \mid & Some(A{\leq}T)\,T & \text{existentially quantified type} \\ & & & \\ e & ::= & \ldots & \\ & \mid & pack\ e\ as\ T\ hiding\ T & \text{packing} \\ & \mid & open\ e\ as\ [A,x]\ in\ e & \text{unpacking} \end{array}$$

The kinding, typing, and subtyping rules are extended as follows:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 \leq T_1 \qquad \Gamma, A{\leq}S_1 \vdash S_2 \leq T_2 \\ \Gamma \vdash Some(A{\leq}S_1)\,S_2 \in \star\end{array}}{\Gamma \vdash Some(A{\leq}S_1)\,S_2 \leq Some(A{\leq}T_1)\,T_2} \qquad \text{(S-Some)}$$

$$\frac{\Gamma, A{\leq}T_1 \vdash T_2 \in \star}{\Gamma \vdash Some(A{\leq}T_1)\,T_2 \in \star} \qquad \text{(K-Some)}$$

$$\frac{\Gamma \vdash T \lesseqgtr Some(A{\leq}U_1)\,U_2 \qquad \Gamma \vdash S \leq U_1 \qquad \Gamma \vdash e \in [S/A]U_2}{\Gamma \vdash pack\ e\ as\ T\ hiding\ S \in T} \qquad \text{(T-Some-I)}$$

$$\frac{\Gamma \vdash e_1 \in Some(A{\leq}S_1)\,S_2 \qquad \Gamma, A{\leq}S_1, x{:}S_2 \vdash e_2 \in T}{\Gamma \vdash open\ e_1\ as\ [A,x]\ in\ e_2 \in T} \qquad \text{(T-Some-E)}$$

The equational theory is extended with the following rules for existential types (c.f. Martin-Löf's weak sigma types):

$$\frac{\Gamma \vdash T \lessgtr Some(A{\le}U_1)\,U_2 \qquad \Gamma \vdash S \le U_1 \qquad \Gamma \vdash e = e' \in [S/A]U2 \qquad \Gamma,\, A{\le}U_1,\, x{:}U_2 \vdash b = b' \in V}{\Gamma \vdash \mathit{open}\ (\mathit{pack}\ e\ \mathit{as}\ T\ \mathit{hiding}\ S)\ \mathit{as}\ [A, x]\ \mathit{in}\ b = [e'/x][S/A]b' \in V} \quad \text{(Eq-Some-Beta)}$$

$$\frac{\Gamma \vdash e \in Some(A{\le}S)\,T}{\Gamma \vdash e = \mathit{open}\ e\ \mathit{as}\ [A, x]\ \mathit{in}\ \mathit{pack}\ x\ \mathit{as}\ Some(A{\le}S)\,T\ \mathit{hiding}\ A \in Some(A{\le}S)\,T} \quad \text{(Eq-Some-Eta)}$$

$$\frac{\Gamma \vdash T \lessgtr Some(A{\le}U_1)\,U_2 \qquad \Gamma \vdash S, S' \le V \le U_1 \qquad \Gamma \vdash e = e' \in [V/A]U_2}{\Gamma \vdash \mathit{pack}\ e\ \mathit{as}\ T\ \mathit{hiding}\ S = \mathit{pack}\ e'\ \mathit{as}\ T\ \mathit{hiding}\ S' \in T} \quad \text{(Eq-Pack)}$$

$$\frac{\Gamma \vdash e = e' \in Some(A{\le}T1)\,T2 \qquad \Gamma,\, A{\le}S_1,\, x{:}S_2 \vdash b = b' \in T}{\Gamma \vdash \mathit{open}\ e\ \mathit{as}\ [A, x]\ \mathit{in}\ b = \mathit{open}\ e'\ \mathit{as}\ [A, x]\ \mathit{in}\ b' \in T} \quad \text{(Eq-Open)}$$

A.8.1. Remark: All of the above rules except for Eq-Some-Eta are valid under the usual encoding of existential types in terms of universal quantifiers:

$$Some(A{\le}S)\,T \overset{\text{def}}{=} All(B)\ (All(A{\le}S)\,T {\to} B) \to B.$$

A.8.2. Definition: The definitions of the predicates co_A and $contra_A$ (Definition A.7.1) are extended as follows:

$$\begin{aligned} co_A(Some(B{\le}T_1)\,T_2) &= A \text{ not free in } T_1 \text{ and } co_A(T_2) \\ contra_A(Some(B{\le}T_1)\,T_2) &= A \text{ not free in } T_1 \text{ and } contra_A(T_2) \end{aligned}$$

A.8.3. Definition: The definitions of the lifting and lowering of a function f through a type T (Definition A.7.2) are extended as follows:

$$\begin{aligned} \mathit{lift}_{Some(B\le T_1)T_2} = \ & \mathit{fun}(g : (Some(B{\le}T_1)\ [X/A]T_2))\\ & \quad \mathit{open}\ g\ \mathit{as}\ [B, x]\ \mathit{in}\\ & \qquad \mathit{pack}\ (\mathit{lift}_{T_2}\ x)\\ & \qquad \mathit{as}\ Some(B{\le}T_1)\ [Y/A]T_2\\ & \qquad \mathit{hiding}\ B\\ & \text{when } A \text{ is not free in } T_1 \end{aligned}$$

$$\begin{aligned} \mathit{lower}_{Some(B\le T_1)T_2} = \ & \mathit{fun}(g : (Some(B{\le}T_1)\ [Y/A]T_2))\\ & \quad \mathit{open}\ g\ \mathit{as}\ [B, x]\ \mathit{in}\\ & \qquad \mathit{pack}\ (\mathit{lower}_{T_2}\ x)\\ & \qquad \mathit{as}\ Some(B{\le}T_1)\ [X/A]T_2\\ & \qquad \mathit{hiding}\ B\\ & \text{when } A \text{ is not free in } T_1 \end{aligned}$$

A.8.4. Remark: Surprisingly, it does not seem to be possible to prove the functoriality laws from the equations Eq-Some-Beta and Eq-Some-Eta. (The difficulty is that, unlike the Eta rules for other type constructors, Eq-Some-Eta does not say that every element of an existential type is in canonical form.) However, if we encode existential types in terms of universal types, then the functoriality of existentials follows from the functoriality of the universal quantifier.

Alternatively, we may obtain functoriality from the intuitively appealing rule

$$\frac{\begin{array}{c}\Gamma \vdash c \in U{\rightarrow}V \qquad \Gamma,\, A{\leq}S_1,\, x{:}S_2 \vdash e_2 \in T \\ \Gamma \vdash e \in \mathit{Some}(A{\leq}S)\,T\end{array}}{c\ (\mathit{open}\ e\ \mathit{as}\ [A,x]\ \mathit{in}\ b) = \mathit{open}\ e\ \mathit{as}\ [A,x]\ \mathit{in}\ (c\ b) \in V} \quad \text{(Eq-Some-Nat)}$$

which, however, does *not* hold in the encoding, unless parametricity is also assumed. (The left- and right-hand sides involve different instantiations of a variable of polymorphic type.) We do not know whether it is possible to derive functoriality from more fundamental equations relating *open* and *pack* that also hold in the encoding.

A.9 Recursive Types

The following extension of the basic $F^\omega_\leq$ calculus with recursive types is somewhat informal. The rules are suggested by existing treatments of recursive types in lower-order calculi [2], but a full study of recursive types in this setting falls outside the scope of the present paper.

The set of F^ω types is extended as follows:

$$\begin{array}{lcll} T & ::= & \ldots & \\ & | & \mu(A{:}K)T & \text{least fixed point} \end{array}$$

The inference rules are extended by the obvious formation rule

$$\frac{\Gamma,\, A{\leq}\mathit{Top}(K) \vdash T \in K}{\Gamma \vdash \mu(A{:}K)T \in K} \quad \text{(K-Mu)}$$

and two subtyping rules — one for "unfolding" a recursive type and one for (finitely) comparing two recursive types (c.f. [2]):

$$\frac{\Gamma \vdash \mu(A{:}K)T \in K}{\Gamma \vdash \mu(A{:}K)T \lesseqgtr [(\mu(A{:}K)T)/A]T} \quad \text{(S-Fold)}$$

$$\frac{\Gamma,\, B{\leq}\mathit{Top}(K),\, A{\leq}B \vdash S \leq T}{\Gamma \vdash \mu(A)S \leq \mu(B)T} \quad \text{(S-Mu)}$$

A.9.1. Remark: Note that we have the derived rule

$$\frac{\begin{array}{l}\Gamma,\, B{\leq}\mathit{Top}(K),\, A{\leq}B,\, D{\leq}\mathit{Top}(K),\, C{\leq}D \\ \qquad\qquad\qquad\qquad \vdash S(A,C) \leq S(B,D)\end{array}}{\Gamma,\, B{\leq}\mathit{Top}(K),\, A{\leq}B \vdash \mu(C)S(A,C) \leq \mu(D)S(B,D)} \quad \text{(S-Mu-Mono)}$$

which states that a type whose outer constructor is μ is monotone in a free variable A if the body of the μ is monotone in both A and the bound variable. This suggests an appropriate extension of the definitions of *co* and *contra*:

A.9.2. Definition: The definitions of the predicates co_A and $contra_A$ (A.7.1) are extended as follows:

$$\begin{array}{lcl} co_A(\mu(B)T_2) & = & co_A(T_2) \text{ and } co_B(T_2) \\ contra_A(\mu(B)T_2) & = & contra_A(T_2) \text{ and } co_B(T_2) \end{array}$$

A.9.3. Remark: A value-level fixed-point combinator

$$\mathit{fix} \in \mathit{All}(A \leq \mathit{Top}(\star))\ (A \rightarrow A) \rightarrow A$$

is definable using recursive types [2].

A.9.4. Definition: The definitions of the lifting and lowering of a function f through a type T (Definition A.7.2) are extended as follows. To reduce clutter, we adopt the abbreviation $S[U,V] = [V/B][U/A]S$.

$$\begin{array}{lll} \mathit{lift}_{\mu(B)S} & = & \mathit{fix}\ [\mu(B)S[X,B] \rightarrow \mu(B)S[Y,B]] \\ & & \quad \mathit{fun}\,(g : \mu(B)\, S[X,B] \rightarrow \mu(B)S[Y,B]) \\ & & \qquad \begin{array}{ll} (\mathit{unfold}: & \mu(B)S[X,B] \rightarrow S[X,\mu(B)S[X,B]]; \\ (\mathit{lift}^{A \leftarrow f}_{S[A,\mu(B)S[X,B]]}: & S[X,\mu(B)S[X,B]] \rightarrow S[Y,\mu(B)S[X,B]]); \\ (\mathit{lift}^{B \leftarrow g}_{S[Y,B]}: & S[Y,\mu(B)S[X,B]] \rightarrow S[Y,\mu(B)S[Y,B]]); \\ (\mathit{fold}: & S[Y,\mu(B)S[Y,B]] \rightarrow \mu(B)S[Y,B]) \end{array} \\ & & \quad \text{when } co_A(S) \text{ and } co_B(S) \end{array}$$

$$\begin{array}{lll} \mathit{lower}_{\mu(B)S} & = & \mathit{fix}\ [\mu(B)S[Y,B] \rightarrow \mu(B)S[X,B]] \\ & & \quad \mathit{fun}\,(g : \mu(B)\, S[Y,B] \rightarrow \mu(B)S[X,B]) \\ & & \qquad \begin{array}{ll} (\mathit{unfold}: & \mu(B)S[Y,B] \rightarrow S[Y,\mu(B)S[Y,B]]; \\ \mathit{lower}^{A \leftarrow f}_{S[A,\mu(B)S[Y,B]]}: & S[Y,\mu(B)S[Y,B]] \rightarrow S[X,\mu(B)S[Y,B]]); \\ (\mathit{lift}^{B \leftarrow g}_{S[X,B]}: & S[X,\mu(B)S[Y,B]] \rightarrow S[X,\mu(B)S[X,B]]); \\ (\mathit{fold}: & S[X,\mu(B)S[X,B]] \rightarrow \mu(B)S[X,B]) \end{array} \\ & & \quad \text{when } contra_A(S) \text{ and } co_B(S) \end{array}$$

(The coercions *fold* and *unfold* and the explicit typings of *lift* and *lower* are included only for the convenience of the reader; in practice, they can be inferred by a typechecker.)

A.9.5. Remark: In order to show that the extended definition of strength still satisfies the functoriality laws (Lemma A.7.6), we need to make some assumptions about the equational theory of the type system with recursive types. In domain-theoretic models of recursive types (and, as far as we know, in metric space models), functoriality holds because the solutions to the recursive equations defining *lift* and *lower* are unique, since the recursion we use defines the iterator of an initial algebra (the recursive types involved are always positive).

B Rules for Objects

B.1 Typing and Subtyping Rules

$$\frac{\Gamma \vdash N \in \star{\to}\star}{\Gamma \vdash \mathit{Object}(N) \in \star} \qquad (\textsc{K-Object})$$

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \\ \Gamma \vdash \mathit{Object}(N') \in \star \qquad \Gamma \vdash \mathit{Object}(N) \in \star\end{array}}{\Gamma \vdash \mathit{Object}(N') \leq \mathit{Object}(N)} \qquad (\textsc{S-Object})$$

$$\frac{\begin{array}{c}\Gamma \vdash N \in \star{\to}\star \qquad co(N) \\ \Gamma \vdash s \in R \qquad \Gamma \vdash m \in R{\to}N(R)\end{array}}{\Gamma \vdash \mathit{object}_N(R, s, m) \in \mathit{Object}(N)} \qquad (\textsc{T-Object-I})$$

$$\frac{\Gamma \vdash N \in \star{\to}\star \qquad co(N)}{\Gamma \vdash GM_N \in \mathit{All}(N'{\leq}N)\ \mathit{Object}(N') \to N(\mathit{Object}(N'))} \qquad (\text{T-GM})$$

B.2 Equational Constraints

$$\begin{tikzcd}
\mathit{Object}(N) \arrow[r, "GM_N\ [N]"] & N(\mathit{Object}(N)) \\
R \arrow[u, "\mathit{object}_N(R,\, —,\, m)"] \arrow[r, "m"'] & N(R) \arrow[u, "\begin{array}{c}\mathit{strength}_N\ [R]\ [\mathit{Object}(N)] \\ \mathit{object}_N(R,\, —,\, m)\end{array}"']
\end{tikzcd} \qquad (\dagger)$$

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \qquad \Gamma \vdash \mathit{Object}(N') \in \star \\ co(N') \qquad co(N) \\ \Gamma \vdash R' \leq R \qquad \Gamma \vdash s \in R' \\ \Gamma \vdash m \in R \to N'(R')\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{object}_{N'}(R', s, m) = \mathit{object}_N(R, s, m) \\ \qquad \in \mathit{Object}(N)\end{array}} \qquad (\textsc{Eq-Object-Nat})$$

$$\frac{\Gamma \vdash N' \leq N \in \star{\to}\star \qquad co(N') \qquad co(N)}{\begin{array}{l}\Gamma \vdash GM_{N'} = GM_N \\ \quad \in \mathit{All}(N''{\leq}N')\ \mathit{Object}(N'') \to N(\mathit{Object}(N''))\end{array}} \qquad (\textsc{Eq-GM-Nat})$$

References

[1] Martin Abadi. Doing without F-bounded quantification. Message to `Types` electronic mail list, February 1992.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, January 1991. Also available as DEC Systems Research Center Research Report number 62, August 1990.

[3] S. Bainbridge, P. Freyd, A. Scedrov, and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[4] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1992. To appear.

[5] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[6] Kim B. Bruce. A paradigmatic object-oriented language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, January 1992.

[7] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[8] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

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

[10] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[11] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished notes, October 1990.

[12] Luca Cardelli. Typed foundations of object-oriented programming. Tutorial given at POPL '92, January 1992.

[13] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, September 1991.

[14] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[15] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[16] W. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker et al., editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

[17] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990.

[18] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *LNCS 389*. Springer-Verlag, 1989.

[19] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. *Mathematical Structures in Computer Science*, 1992. To appear.

[20] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[21] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[22] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[23] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA '86 Conference Proceedings, Portland, Oregon*, pages 347–349, September 1986. Appeared as a special issue of SIGPLAN Notices 21, 11, November, 1986.

[24] A. Kock. Strong functors and monoidal monads. *Various Publications Series 11, Aarhus Universitet*, 1970.

[25] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[26] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990.

[27] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming Series, pages 195–212. Addison-Wesley, 1990.

[28] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[29] P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, pages 217–238. Cambridge University Press, Cambridge, 1992.

[30] Benjamin C. Pierce and Robert Pollack. Higher-order subtyping. Unpublished manuscript, August 1992.

[31] Benjamin C. Pierce and David N. Turner. Binary methods. Draft technical teport, August 1992.

[32] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. Technical Report ECS-LFCS-92-225, University of Edinburgh, LFCS, 1992.

[33] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. Draft report, August 1992.

[34] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 289–297, Snowbird, Utah, July 1988.

[35] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[36] John C. Reynolds. User defined types and procedural data structures as complementary approaches to data abstraction. In David Gries, editor, *Programming Methodology, A Collection of Articles by IFIP WG2.3*, pages 309–317. Springer-Verlag, New York, 1978. Reprinted from S. A. Schuman (ed.), *New Advances in Algorithmic Languages 1975*, Inst. de Recherche d'Informatique et d'Automatique, Rocquencourt, 1975, pages 157-168.

[37] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[38] C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.

[39] B. Stroustrup. *C++*. Addison-Wesley, Reading, Mass, 1986.

[40] Gavin C. Wraith. A note on categorical datatypes. Number 389 in Lecture Notes in Computer Science. Springer-Verlag, 1989.