J. Functional Programming **1** (1): 1–000, January 1993

A Unifying Type-Theoretic Framework for Objects

Martin Hofmann Benjamin Pierce

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

Abstract

We give a direct type-theoretic characterization of the basic mechanisms of object-oriented programming, including objects, methods, message passing, and subtyping, by introducing an explicit constructor for object types and suitable introduction, elimination, and equality rules. The resulting abstract framework provides a basis for justifying and comparing previous encodings of objects based on recursive record types (Cardelli, 1984; Cardelli, 1992; Bruce, 1994; Cook *et al.*, 1990; Mitchell, 1990a) and encodings based on existential types (Pierce & Turner, 1994).

1 Introduction

Research on the foundations of object-oriented programming languages has produced a series of increasingly ambitious attempts to capture the static typing properties of well-behaved programs in conventional object-oriented languages (Cardelli, 1984; Wand, 1987; Cook *et al.*, 1990; Mitchell, 1990a; Cardelli, 1992; Bruce, 1994; Castagna *et al.*, 1994; Pierce & Turner, 1994; Abadi, 1994; Abadi & Cardelli, 1994a; Abadi & Cardelli, 1994b; Fisher & Mitchell, 1994, etc.). These proposals have often focused on encodings of high-level syntax for objects into more primitive constructions in various typed λ-calculi, the semantics of objects being understood simply as the semantics of their encodings. Our goal here is to use the tools of type theory to give a more direct account of objects and message passing, with the aim of isolating high-level principles for reasoning about objects.

An *object* in the sense of Smalltalk (Goldberg & Robson, 1983) can be thought of as a *state* of some hidden *representation type* together with a collection of *methods* that are used to analyze or change the state. For example, take simple one-dimensional point objects with the operations `set` and `get`, where `set` is expressed in a functional style, returning an object with an updated state rather than modifying the state in-place. Such objects can be implemented by choosing a representation type, say `Int`, a state, say `5`, and two method implementations, say

```
set = fun(state:Int) fun(newX:Int) newX
get = fun(state:Int) state
```

and packaging them together so that the state is protected from external access except via the methods. Because we are mainly concerned here with the typing properties of object-oriented features, this very simple example will suffice for our purposes throughout the article. The applicability of these techniques to larger examples and to objects with mutable state are discussed in Pierce and Turner (1994).

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 `set` and `get` methods, appropriate to its internal representation type. Thus, another implementation of point objects might use the more interesting representation type `{x:Int,other:Int}`, the initial state `{x=5,other=8}`, and the following methods:

```
set = fun(state:{x:Int,other:Int}) fun(i:Int) {x=i, other=state.other}
get = fun(state:(x:Int,other:Int}) state.x
```

This flexibility is central to the spirit of object-oriented programming. Although it is not found in its most general form in some object-oriented languages (e.g. Smalltalk and C++ (Stroustrup, 1986), which do not allow multiple implementations of a class), equivalent mechanisms like "virtual classes" are then used in its place. The crucial point is that when a message is sent to an object, the identity of the object itself determines what code is executed in response. Thus, 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 uses a uniform function

```
Point'set : Point -> Int -> Point
```

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

```
Point'get : Point -> Int
```

that uniformly invokes the internal `get` method of any point and returns the resulting integer. One of our goals below will be to define the word "uniform" rigorously. Intuitively, it means that the behavior of a message-sending function is completely determined by the implementation of the corresponding method of the object; in particular, a message-sending function cannot involve a "typecase" on its argument's representation.

Our purpose is to study the mechanisms of encapsulation and message passing in a type-theoretic setting. In Sections 2 and 3, 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 4, we extend this framework to include subtyping. Section 5 justifies the framework by showing that a simple encoding of objects in terms of existential types satisfies our axioms. Section 6 shows that the more familiar encoding of object types as recursive records also satisfies the axioms; Section 7 discusses the special case of F-bounded quantification. In Section 8 we use our abstract framework to sketch a high-level concrete syntax for object type declarations and message passing operations. Section 9 extends the framework to mixed-variance method signatures, illustrating the correspondence between our approach and previously studied encodings of mixed-variance objects; here, recursive types turn out to be unavoidable. Section 10 offers concluding remarks.

Appendices A through E develop the formal foundations of the type theories used in the body of the article. We begin in Appendix A with the typed λ-calculus $F^\omega_\leq$ (an extension of Girard's System F^ω with subtyping), reviewing the standard typing and subtyping rules and presenting a new equational theory generalizing the one developed by Cardelli, Martini, Mitchell, and Scedrov (1994) for $F_\leq$, the second-order fragment of $F^\omega_\leq$. Following the informal development in Section 3, Appendix B extends $F^\omega_\leq$ with a predicate *pos* for testing the positivity of type operators and a polymorphic constant *map* that can be used to "map a given function through a positive operator." Appendix C summarizes the typing, subtyping, and equational rules for the *Object* type constructor and its associated term constructors. (These rules are introduced, by a series of refinements, in the body of the article. The purpose of Appendix C is to collect the final versions in a single place.) Appendices D and E describe extensions of $F^\omega_\leq$ with existential types and recursive types. For existential types we offer an equational theory, which can be shown to be sound in a standard PER model and which satisfies the laws in Appendix C. The extension with recursive types is more problematic: it appears difficult to give a purely equational axiomatization from which, for example, the laws governing *map* in Appendix B can be derived. Instead, we offer an argument from semantic considerations.

Our presentation is self-contained. However, basic familiarity with Girard's F^ω, type systems with subtyping, other type-theoretic treatments of objects, and basic terminology of category theory will be helpful. Background reading in these areas can be found in (Barendregt, 1992; Pierce *et al.*, 1989; Cardelli *et al.*, 1994; Barr & Wells, 1990; Fisher & Mitchell, 1994) and elsewhere in the references.

2 Motivation

The representation type of an object is hidden from external view: its *interface* is just the types of its methods. Type-theoretically, the interface 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) {set: X->Int->X, get: X->Int}.
```

For now, we leave unspecified the "ambient type theory" in which our definitions are embedded. In Sections 5 and 6, we will be using two extensions of the higher-order polymorphic λ-calculus System F^ω (Girard, 1972) with subtyping; these are summarized in the appendices. The calculus under consideration will determine the precise force of the equational constraints expressed by the diagrams.

To characterize the set of objects sharing a common interface, we introduce a new type constructor *Object*, which turns an interface specification (a type operator of kind *Type*→*Type*, i.e. a map from types to types) into a type:

$$\frac{\Gamma \vdash M \ : \ Type \rightarrow Type}{\Gamma \vdash Object(M) \ : \ Type} \qquad \text{(K-Obj*)}$$

(We will alter this rule slightly later on; final versions of all the rules are given in Appendix C. Rules that will be superseded by others appearing later are given names ending with a *.) 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 methods, and an initial value s of the representation type, we use *object* to package them together into an element $object_M\,(R,\, s,\, m) \ : \ Object(M)$. For example, two point objects with different representation types can be created as follows:

```
m1 = {set = fun(state:Int) fun(i:Int) i, get = fun(state:Int) state}
   : PointM(Int)

p1 = object_PointM(Int, 5, m1)
   : Point

m2 = {set = fun(state:{x:Int,other:Int})
              fun(i:Int) {x=i, other=state.other},
      get = 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.

$$\frac{\Gamma \vdash M \ : \ Type \rightarrow Type \qquad \Gamma \vdash s \ : \ R \qquad \Gamma \vdash m \ : \ M(R)}{\Gamma \vdash object_M\,(R,\, s,\, m) \ : \ Object(M)} \qquad \text{(T-Obj-I*)}$$

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 (1988). 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 want to capture this mechanism directly.

Since every point object must implement the `set` method, there should be a uniform function

```
Point'set : Point -> Int -> Point
```

that, given a point, invokes its `set` method. More generally, for each operator M representing the interface of an object type, we introduce a term constant GM_M denoting the whole collection of uniform message-sending functions corresponding to this signature.

$$\frac{\Gamma \vdash M \ : \ Type \rightarrow Type}{\Gamma \vdash GM_M \ : \ M(Object(M))} \qquad \text{(T-GM*)}$$

We call this the "generic method" for objects with interface M. Then

```
GM_Point : {set: Point->Int->Point, get: Point->Int}
Point'set = GM_Point.set.
```

Such uniform method invocation functions do not necessarily exist. For example, we might extend our first implementation of points with an equality method

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

(where `eqInt : Int->Int->Bool` is the equality function on integers), but we cannot expect to be able to invoke this `eq` uniformly — that is, we cannot expect to write a function

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

that calls the `eq` method of its first parameter and passes it the internal representations of both parameters: such an invocation of the low-level `eq` function would only be well typed when the two points passed as arguments to `Point'eq` happen to have *identical* representation types, which is in general not the case.

This example illustrates a well-known, inherent limitation of object-style encapsulation (Reynolds, 1978; Cook, 1991). 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 at a time. This limitation can be modeled abstractly as a syntactic restriction on M, capturing the intuition that the methods should all be 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. For example, we can express the signature of points as

```
Fun(R) R -> PointN(R)
```

where `PointN = Fun(R) {set: Int->R, get: Int}`. The restriction to a single record-valued method is purely a matter of formal convenience; in practice, one could allow several methods here.

Formally, 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. For the case of $F^{\omega}_{\leq}$, positive occurrences are defined in Appendix B by induction on the structure of types. We write $pos(N)$ in formulas to assert that the parameter A only appears positively in the body T of the operator $N = \mathit{Fun}(A)\,T$. Such operators are called *positive*.

3 Objects

We will henceforth restrict our attention to objects with unary methods, using the positive operator N rather than $M = \mathit{Fun}(R)\ \ R \rightarrow N(R)$ as the parameter to the

Object type constructor. This entails a small modification to our typing rules for the *object* constructor and the generic method:

$$\frac{\Gamma \vdash N \ :\ Type \rightarrow Type}{\Gamma \vdash Object(N) \ :\ Type} \qquad \text{(K-OBJ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash N \ :\ Type \rightarrow Type \qquad pos(N) \\ \Gamma \vdash s \ :\ R \qquad \Gamma \vdash m \ :\ R \rightarrow N(R)\end{array}}{\Gamma \vdash object_N\,(R,\, s,\, m) \ :\ Object(N)} \qquad \text{(T-OBJ-I)}$$

$$\frac{\Gamma \vdash N \ :\ Type \rightarrow Type \qquad pos(N)}{\Gamma \vdash GM_N \ :\ Object(N) \rightarrow N(Object(N))} \qquad \text{(T-GM**)}$$

It might seem cleaner here to allow the formation of $Object(N)$ only when $pos(N)$. But formulating the rules in this way would require, in Section 4, that we be able to quantify over positive operators, since there we will need to consider object types of the form $Object(N)$ where N is a variable. The study of a refined type theory in which this would be possible is an interesting topic for future research, but using such a type theory would significantly complicate our formal development without adding much insight.

Next, we need 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 of applying the generic method to a newly built object should be the same as the result of applying the concrete method to the object's state and then "repacking" the result — i.e., the informal diagram

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

should commute, where $object_N\,(R, —, m)$ stands for the packing function $fun(x : R)\ object_N\,(R, x, m)$. Note that, for any given R and m, this specification can be satisfied trivially by setting $Object(N) = R$ and $GM_N = m$; the force of the diagram lies in the fact that $Object(N)$ and GM_N independent of R — a consequence of the typing rules.

In the case of our simpler implementation of points, this diagram can equivalently be written as a pair of equations

```
  (GM_PointN (object_PointN(Int,s,m))).set(i)
= object_PointN(Int, (m s).set i, m)

  (GM_PointN (object_PointN(Int,s,m))).get
= s,
```

corresponding to the `set` and `get` methods, respectively, where

```
m = fun(s:Int) {set = fun(i:Int) i, get = s}.
```

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

```
fun(r: {set:Int->Int,get:Int})
  {set = fun(i:Int) object_PointN(Int, r.set i, m),
   get = r.get}.
```

This special case brings us to a technical cornerstone of the article: the observation that, in the general case, this repacking function can be expressed using the idea of "mapping a function through a positive type operator."

In higher-order polymorphic λ-calculi like F^ω and $F^\omega_\leq$, the action of a positive type operator N on a function $f : X \rightarrow Y$ can be interpreted as "applying f to each occurrence of X in $N(X)$" — that is, given an element $n : N(X)$, decompose n, apply f to each component of type X, and use the results to rebuild an element of $N(Y)$. For example, if $N(X) = \{a{:}X,\ b{:}Int,\ c{:}Bool \rightarrow X\}$, this procedure yields

$$\begin{array}{rl} & \{a = f(n.a),\ b = n.b,\ c = fun\,(v{:}Bool)\, f(n.c\ v)\} \\ : & \{a{:}Y,\ b{:}Int,\ c{:}Bool \rightarrow Y\} \\ = & N(Y) \end{array}$$

We will henceforth assume that the ambient type theory supports a predicate $pos(N)$ on elements N of $Type \rightarrow Type$ and an appropriate function map_N for every N with $pos(N)$. Appendices B, D, and E show how $pos(N)$ and map_N can be defined for the type theories under consideration.

3.1 Definition: A type theory extending F^ω is said to *include positivity* if it provides a predicate *pos* on operators of kind $Type \rightarrow Type$ and, for each N with $pos(N)$, there is a polymorphic function

$$map_N \ :\ All(X)\, All(Y)\ \ (X \rightarrow Y) \rightarrow (N(X) \rightarrow N(Y))$$

such that

$$\frac{\begin{array}{c}\Gamma \vdash N \ :\ Type \rightarrow Type \qquad pos(N) \\ \Gamma \vdash f \ :\ X \rightarrow Y \qquad \Gamma \vdash g \ :\ Y \rightarrow Z\end{array}}{\begin{array}{l}\Gamma \vdash map_N\ [X]\ [Z]\ (f;g) \\ \quad = (map_N\ [X]\ [Y]\ f)\ ;\ (map_N\ [Y]\ [Z]\ g) \\ \quad :\ N(X) \rightarrow N(Z)\end{array}} \qquad \text{(Eq-Map-Trans)}$$

$$\frac{\Gamma \vdash N \ :\ Type \rightarrow Type \qquad pos(N) \qquad \Gamma \vdash X \ :\ Type}{\Gamma \vdash map_N\ [X]\ [X]\ (id\ [X]) = id\ [N(X)]\ :\ N(X) \rightarrow N(X)} \qquad \text{(Eq-Map-Id)}$$

where $f;g = \lambda x.\ g(f(x))$ denotes composition in diagrammatic order and id is the polymorphic identity function.

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

$$\begin{tikzcd}
Object(N) \arrow[r, "GM_N"] & N(Object(N)) \\
R \arrow[u, "object_N\,(R,\, —,\, m)"] \arrow[r, "m"'] & N(R) \arrow[u, "map_N\ [R]\ [Object(N)]\ object_N\,(R,\, —,\, m)"']
\end{tikzcd}$$

(Eq-Obj-Map*)

commute, or equivalently that the following equation be satisfied:

$$GM_N(object_N\,(R,\, s,\, m)) = map_N\ [R]\ [Object(N)]\ (object_N\,(R,\, —,\, m))\ (m\ s)$$
$$:\ N(Object(N)).$$

We close this section with several technical remarks. First, it is interesting to note that by orienting this equation from left to right, we obtain a natural computation rule for objects. This suggests that a different kind of semantics for our abstract calculus — besides those developed in Sections 5 and 6 — could be obtained by adding this reduction to a standard operational semantics for $F^{\omega}_{\leq}$.

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

$$\frac{\Gamma \vdash x\ :\ Object(N)}{\Gamma \vdash x\ =\ object_N\,(Object(N),\, x,\, GM_N)\ :\ Object(N)}$$

We prefer to regard this axiom as optional, since it places a strong constraint on the encodings we discuss below.

Our *map* operator arises from the concept of functorial strength in category theory (Kock, 1970; Moggi, 1989): an endofunctor $N\ :\ \mathbf{C} \rightarrow \mathbf{C}$ 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 $map_{X,Y} : (X \Rightarrow Y) \rightarrow (N(X) \Rightarrow N(Y))$ that captures the action of N.

Finally, it is interesting to note that our abstract specification of objects exactly amounts to defining $Object(N)$ 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 (1989). 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 coarser equality for objects given by bisimulation equivalence. This intuition also underlies the categorical approach to object semantics proposed by Reichel (1995).

4 Objects and Subtyping

Next, we extend our abstract characterization of objects and message passing to include another important concept from object-oriented programming languages: subtyping. The issues we must deal with are as follows:

1. The λ-calculus in which the model is expressed must be extended with subtyping.
2. The subtyping behavior of the *Object* type constructor must be specified. For example, if we introduce a type of colored point objects whose interface includes `setC` and `getC` methods in addition to `set` and `get`, 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 System F^{ω} with subtyping have been proposed (Cardelli, 1990; Bruce & Mitchell, 1992; Pierce & Turner, 1994; Compagnoni & Pierce, 1993; Steffen & Pierce, 1994; Compagnoni, 1994); we choose the simplest (Steffen & Pierce, 1994). The formulation of this system, called $F^{\omega}_{\leq}$, follows the pattern used by Cardelli and Wegner to obtain $F_{\leq}$ from the pure polymorphic λ-calculus (Girard, 1972; Reynolds, 1974):

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

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e : T}$$

- For each kind K, we add a maximal element $Top(K)$.
- 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 $Fun(A{:}K)\,T$ rather than changing to $Fun(A{\leq}S)\,T$. 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 $A{\leq}\,Top(K)$ instead:

$$\frac{\Gamma,\, A{\leq}\,Top(K_1) \vdash T_2 \;:\; K_2}{\Gamma \vdash Fun(A{:}K_1)\,T_2 \;:\; K_1{\rightarrow}K_2}$$

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

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

  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 $U : K$.

- Because subtyping of operators is pointwise, we may promote the operator in a type application to any larger operator "in place":

$$\frac{\Gamma \vdash S \leq T}{\Gamma \vdash S\ U \leq T\ U}$$

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

The resulting calculus is summarized in Appendix A.

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

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \qquad \mathit{pos}(N') \qquad \mathit{pos}(N) \\ \Gamma \vdash f : X{\rightarrow}Y \qquad \Gamma \vdash n : N'(X)\end{array}}{\Gamma \vdash map_{N'}\ [X]\ [Y]\ f\ n = map_N\ [X]\ [Y]\ f\ n : N(Y)} \qquad \text{(Eq-Map-Sub)}$$

i.e.,

$$\begin{tikzcd}
N(X) \arrow[rr, "map_N\ [X]\ [Y]\ f"] & & N(Y) \\
N'(X) \arrow[u, "\leq"] \arrow[rr, "map_{N'}\ [X]\ [Y]\ f"'] & & N'(Y) \arrow[u, "\leq"']
\end{tikzcd}$$

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

Moreover, we require that all positive operators be monotone with respect to the subtyping relation:

$$\frac{\Gamma \vdash N : \mathit{Type}{\rightarrow}\mathit{Type} \qquad \mathit{pos}(N) \qquad \Gamma \vdash S \leq T}{\Gamma \vdash N\,S \leq N\,T} \qquad \text{(S-Pos-Mono)}$$

In Appendices B, D, and E we show that Eq-Map-Sub and S-Pos-Mono hold for the particular definitions of $\mathit{pos}(N)$ and map_N exhibited there.

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

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

where

```
CPointN = Fun(X) {set: Int->X, get: Int, setC: Color->X, getC: Color}.
```

This leads to the following subtyping rule for object types:

$$\frac{\Gamma \vdash N' \leq N \qquad \Gamma \vdash N \ : \ \mathit{Type} \rightarrow \mathit{Type}}{\Gamma \vdash \mathit{Object}(N') \leq \mathit{Object}(N)} \qquad (\text{S-OBJ})$$

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 `set` method 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 positive operator and $N' \leq N$. The application of GM_N to an element of $\mathit{Object}(N')$ should yield an element of $N(\mathit{Object}(N'))$, not $N(\mathit{Object}(N))$ as above. This suggests a change in the type of GM:

$$\frac{\Gamma \vdash N \ : \ \mathit{Type}{\rightarrow}\mathit{Type} \qquad \mathit{pos}(N)}{\Gamma \vdash GM_N \ : \ \mathit{All}(N'{\leq}N) \ \ \mathit{Object}(N') \rightarrow N(\mathit{Object}(N'))} \qquad (\text{T-GM})$$

(c.f. (Cardelli & Wegner, 1985)). Note, here, that $N'{\leq}N$ does not imply that N' is also positive.

When GM_N is applied to N itself, the original specification should continue to hold:

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

(EQ-OBJ-MAP)

or, as an equational rule:

$$\frac{\begin{array}{c}\Gamma \vdash N \ : \ \mathit{Type}{\rightarrow}\mathit{Type} \qquad \mathit{pos}(N) \\ \Gamma \vdash s \ : \ R \qquad \Gamma \vdash m \ : \ R{\rightarrow}N(R)\end{array}}{\begin{array}{l}\Gamma \vdash GM_N \ [N] \ (\mathit{object}_N\,(R,\, s,\, m)) \\ \quad = \mathit{map}_N \ [R] \ [\mathit{Object}(N)] \ (\mathit{object}_N\,(R,\, \text{---},\, m)) \ (m \ s) \\ \quad : \ N(\mathit{Object}(N))\end{array}} \qquad (\text{EQ-OBJ-MAP})$$

In fact, the examples below show that this special instance of the commutativity of *object* and *map* actually constrains their behavior in a much broader range of situations. We take this diagram as a basic axiom.

The interaction between the subtype relation and the term constructors *object* and *GM* is axiomatized by two rules like EQ-MAP-SUB, which stipulate that they should commute when all the operators involved are positive.

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \qquad \mathit{pos}(N') \qquad \mathit{pos}(N) \\ \Gamma \vdash s \ : \ R \qquad \Gamma \vdash m \ : \ R \rightarrow N'(R)\end{array}}{\Gamma \vdash \mathit{object}_{N'}\,(R,\, s,\, m) = \mathit{object}_N\,(R,\, s,\, m) \ : \ \mathit{Object}(N)} \qquad (\text{EQ-OBJ-SUB})$$

$$\frac{\Gamma \vdash N'' \leq N' \leq N \;:\; \mathit{Type}{\to}\mathit{Type} \qquad \mathit{pos}(N') \qquad \mathit{pos}(N)}{\Gamma \vdash \mathit{GM}_{N'}\;[N''] = \mathit{GM}_N\;[N''] \;:\; \mathit{Object}(N''){\to}N(\mathit{Object}(N''))} \quad (\textsc{Eq-GM-Sub})$$

(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. (Cardelli *et al.*, 1994; Mitchell, 1990b).)

4.1 Example: Mitchell's treatment of method specialization and inheritance via natural transformations (1990a) includes a "coherence condition" between different instances of a given generic method. If $N'' \leq N' \leq N$ and $\mathit{pos}(N)$, then:

$$\begin{array}{ccc} \mathit{Object}(N') & \xrightarrow{\mathit{GM}_N[N']} & N(\mathit{Object}(N')) \\ \leq \Big\uparrow & & \Big\uparrow \leq \\ \mathit{Object}(N'') & \xrightarrow[\mathit{GM}_N[N'']]{} & N(\mathit{Object}(N'')) \end{array}$$

The commutativity of this diagram also follows from our laws.

Proof: (This proof and those that follow depend on definitions and results from the appendices. Readers who wish to follow in detail should first familiarize themselves with the material presented there.)

By T-GM and Eq-Refl from Appendix A.8,

$$\Gamma \vdash \mathit{GM}_N = \mathit{GM}_N \;:\; \mathit{All}(N'{\leq}N)\;\; \mathit{Object}(N') \to N(\mathit{Object}(N')).$$

On the other hand, by S-Object, S-Arrow, and S-Pos-Mono,

$$\Gamma \vdash \mathit{Object}(N') \to N(\mathit{Object}(N')) \leq \mathit{Object}(N'') \to N(\mathit{Object}(N'))$$

and

$$\Gamma \vdash \mathit{Object}(N'') \to N(\mathit{Object}(N'')) \leq \mathit{Object}(N'') \to N(\mathit{Object}(N')).$$

Eq-TApp now applies, yielding

$$\Gamma \vdash \mathit{GM}_N\;[N'] = \mathit{GM}_N\;[N''] \;:\; \mathit{Object}(N'') \to N(\mathit{Object}(N')),$$

as required. □

4.2 Example: Eq-GM-Sub can be used to derive a similar kind of coherence between *different* generic methods. If $N' \leq N$ with $\mathit{pos}(N)$ and $\mathit{pos}(N')$, then:

$$\begin{array}{ccc} \mathit{Object}(N) & \xrightarrow{\mathit{GM}_N[N]} & N(\mathit{Object}(N)) \\ \leq \Big\uparrow & & \Big\uparrow \leq \\ \mathit{Object}(N') & \xrightarrow[\mathit{GM}_{N'}[N']]{} & N'(\mathit{Object}(N')) \end{array}$$

Proof: Use Eq-GM-Sub with $N'' = N'$ to obtain

$$\Gamma \vdash GM_{N'}\ [N'] = GM_N\ [N']\ :\ Object(N') \rightarrow N(Object(N')),$$

which, by Eq-Subsumption (using S-Pos-Mono), gives

$$\Gamma \vdash GM_{N'}\ [N'] = GM_N\ [N']\ :\ Object(N') \rightarrow N(Object(N)).$$

Now rename N' to N and N'' to N' in the diagram from Example 4.1, yielding

$$\Gamma \vdash GM_N\ [N] = GM_N\ [N']\ :\ Object(N') \rightarrow N(Object(N)).$$

The desired result follows by symmetry and transitivity. □

4.3 Example: Similarly, we can combine Eq-GM-Sub and Eq-Obj-Map to characterize the behavior of the generic method when applied to some refinement N' of its own interface operator N:

$$\begin{tikzcd}
Object(N') \arrow[r, "GM_N[N']"] & N(Object(N')) \\
 & N'(Object(N')) \arrow[u, "\leq"'] \\
R \arrow[uu, "object_{N'}(R,\, —,\, m)"] \arrow[r, "m"'] & N'(R) \arrow[u, "\begin{array}{l} map_{N'}\ [R]\ [Object(N')] \\ object_{N'}(R,\, —,\, m) \end{array}"']
\end{tikzcd}$$

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 (Pierce & Turner, 1994; Läufer & Odersky, 1994) (c.f. Danforth and Tomlinson (1988) and Bruce (1994); the latter is mainly based on recursive records, but existential types are used to implement hidden instance variables). The type $Object(N)$ is defined as an abstract type in the sense of Mitchell and Plotkin (1988), with hidden representation A, a state of type A, and an implementation of the methods of type $A \rightarrow N(A)$

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

or, in 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 D, following Cardelli and Wegner (1985) and Mitchell and Plotkin (1988). The definition of *map* for this calculus, a straightforward extension of the definition of *map* for pure $F^{\omega}_{\leq}$, is also given in Appendix D.

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

$$\begin{array}{lcl}
Object(N) & = & Some(A)\ \{\ state : A,\ methods : A \rightarrow N(A)\ \ \} \\[1em]
object_N\,(R,\ s,\ m) & = & pack\ \{state = s,\ methods = m\}\ as\ Object(N) \\
 & & hiding\ R \\[1em]
GM_N & = & fun(N' {\leq} N) \\
 & & \quad fun(x : Object(N')) \\
 & & \qquad open\ x\ as\ [R,\ r]\ in \\
 & & \qquad\quad map_N\ [R]\ [Object(N')] \\
 & & \qquad\qquad object_{N'}\,(R,\ \text{—},\ r.methods) \\
 & & \qquad\qquad (r.methods\ r.state)
\end{array}$$

5.2 Proposition: This encoding satisfies the object axioms summarized in Appendix C.

Proof: The typing and subtyping laws, K-Obj, S-Obj, T-Obj-I, and T-GM, follow directly from the definitions. The three equational laws are more interesting.

Eq-Obj-Map follows by the rules Eq-TBeta, Eq-Beta, Eq-Abs, and Eq-Some-Beta.

Eq-GM-Sub follows by Eq-Map-Sub, Eq-Open, Eq-Abs, Eq-Subsumption, and Eq-TBeta.

Eq-Obj-Sub is derived as follows. We are given

$$\begin{array}{l}
\Gamma \vdash N' \leq N \\
pos(N') \qquad pos(N) \\
\Gamma \vdash s\ :\ R \\
\Gamma \vdash m\ :\ R \rightarrow N'(R).
\end{array}$$

Let

$$\begin{array}{lcl}
V & = & \{state : R,\ methods : R \rightarrow N'(R)\} \\
body & = & \{state = s,\ methods = m\}.
\end{array}$$

Then

$$\Gamma \vdash body = body\ :\ V$$

by Eq-Refl. By S-App, S-Arrow, and S-Rcd,

$$\begin{array}{l}
\Gamma \vdash V \leq \{state : R,\ methods : R \rightarrow N(R)\} \\
\Gamma \vdash V \leq \{state : R,\ methods : R \rightarrow N'(R)\}.
\end{array}$$

Now, by Eq-Pack,

$$\begin{array}{rl}
\Gamma\ \vdash & pack\ body\ as\ Some(A)\,\{state : A,\ methods : A \rightarrow N'(A)\}\ hiding\ R \\
= & pack\ body\ as\ Some(A)\,\{state : A,\ methods : A \rightarrow N(A)\}\ hiding\ R \\
: & Some(A)\,\{state : A,\ methods : A \rightarrow N(A)\},
\end{array}$$

i.e.,

$$\Gamma \vdash object_{N'}\,(R,\ s,\ m) = object_N\,(R,\ s,\ m)\ :\ Object(N). \qquad \square$$

This encoding is interesting both because it works in a fairly simple calculus — pure $F^{\omega}_{\leq}$ enriched with existential types — and because it avoids introducing the possibility of non-termination in situations where fixed points are not strictly required. One situation where fixed points at the value level *are* required is the modeling of inheritance, where the pseudovariable `self` is given meaning by taking a fixed point of a method-building function; a more detailed discussion of this point can be found in (Pierce & Turner, 1994; Hofmann & Pierce, 1994).

6 Objects as Recursive Records

Next, we show that a familiar encoding of objects as recursive records (Cardelli, 1984; Cardelli, 1992; Bruce, 1994; Mitchell, 1990a, etc.) satisfies the specification developed in Sections 3 and 4 and summarized in Appendix C. This justifies both the abstract framework itself (by showing that a well-known 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 pure $F^{\omega}_{\leq}$ with a recursive type constructor μ, which obeys the following subtyping laws (Amadio & Cardelli, 1993):

$$\frac{\Gamma \vdash \mu(A)T \ : \ \mathit{Type}}{\Gamma \vdash \mu(A)T \sim [(\mu(A)T)/A]T} \qquad \text{(S-FOLD*)}$$

$$\frac{\Gamma,\ B{\leq}\,\mathit{Top}(\mathit{Type}),\ A{\leq}B \vdash S \leq T}{\Gamma \vdash \mu(A)S \leq \mu(B)T} \qquad \text{(S-MU*)}$$

This extension is summarized in Appendix E. (We give slightly more general versions of S-FOLD and S-MU there.) We also assume the existence of a fixed-point combinator

$$\mathit{fix}\ :\ \mathit{All}(A{\leq}\,\mathit{Top}(\mathit{Type}))\ \ (A{\rightarrow}A) \rightarrow A.$$

This combinator can be defined using mixed-variance μ-types (Amadio & Cardelli, 1993); however, we prefer to consider it as a primitive, since the equation EQ-FIX-SUB in Appendix E is not provable syntactically for the encoding. The definitions of positivity and *map* for the extended system appear in Appendix E.

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

$$\mathit{Object}(N) \quad = \quad \mu(X)\ N(X),$$

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 in Appendix A.8, that the extension of a function is its input-output behavior.

Now, $\mathit{Object}(N)$ is both a sub- and a supertype of $N(\mathit{Object}(N))$ by S-MU. Therefore, the generic method can be implemented as an identity function:

$$\begin{array}{rcl}
\mathit{GM}_N & = & \mathit{fun}(N'{\leq}N)\ \ \mathit{id}\ [\mathit{Object}(N')] \\
 & : & \mathit{All}(N'{\leq}N)\ \ \mathit{Object}(N') \rightarrow \mathit{Object}(N') \\
 & \sim & \mathit{All}(N'{\leq}N)\ \ \mathit{Object}(N') \rightarrow N'(\mathit{Object}(N')) \\
 & \leq & \mathit{All}(N'{\leq}N)\ \ \mathit{Object}(N') \rightarrow N(\mathit{Object}(N')).
\end{array}$$

The top arrow $GM_N\ [N]$ in the diagram corresponding to rule Eq-Obj-Map is now invertible. Therefore, Eq-Obj-Map is satisfied iff

$$\begin{array}{l}\Gamma \vdash \mathit{object}_N\,(R,\, s,\, m)\\ \quad = \mathit{map}_N\ [R]\ [\mathit{Object}(N)]\ (\mathit{object}_N\,(R,\, \text{—},\, m))\ (m\ s)\\ \quad :\ \ \mathit{Object}(N)\end{array}$$

where

$$\Gamma \vdash N\ :\ \mathit{Type}{\rightarrow}\mathit{Type} \qquad \mathit{pos}(N) \qquad \Gamma \vdash s\ :\ R \qquad \Gamma \vdash m\ :\ R{\rightarrow}N(R).$$

In other words, the diagram can be read as a recursive specification of the *object* constructor, which can be solved using the fixed point combinator:

$$\mathit{object}_N\,(R,\, s,\, m) \quad = \quad \mathit{obj}\ s$$

where

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

(The function *fold* is actually an implicit coercion; we write it explicitly here as an aid to the reader.)

For example, suppose we are given the representation type `Int` and the following implementation of the point methods:

```
m = fun(s:Int) {get = s, set = fun(i:Int) i}
  : Int -> PointN(Int)
```

Then by expanding the definitions of `object` and `map_PointN` and β-reducing, 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 {get = s, set = fun(i:Int) mkp i})
```

6.1 Proposition: The object laws in Appendix C are satisfied by this encoding.

Proof: The kinding, subtyping, and typing rules are established by straightforward calculation, using S-Pos-Mono for the case of S-Obj. As in Proposition 5.2, the equational rules are more interesting.

For Eq-Obj-Sub, we are given

$$\begin{array}{l}\Gamma \vdash N' \leq N\\ \mathit{pos}(N') \qquad \mathit{pos}(N)\\ \Gamma \vdash s\ :\ R\\ \Gamma \vdash m\ :\ R \rightarrow N'(R).\end{array}$$

From Eq-Map-Sub and Eq-TApp, we obtain

$$\begin{array}{rl}\Gamma,\, f{:}R{\rightarrow}\mathit{Object}(N'),\, s{:}R \vdash & \mathit{map}_{N'}\ [R]\ [\mathit{Object}(N')]\ f\ (m\ s)\\ & = \mathit{map}_N\ [R]\ [\mathit{Object}(N)]\ f\ (m\ s)\\ & :\ N(\mathit{Object}(N)).\end{array}$$

Using EQ-ABS and EQ-ABS$^+$ (A.8.2), we deduce

$$\begin{aligned}\Gamma \vdash\ & \mathit{fun}(f{:}R{\rightarrow}\mathit{Object}(N'))\ \ \mathit{fun}(s{:}R)\ \mathit{map}_{N'}\ [R]\ [\mathit{Object}(N)']\ f\ (m\ s)\\ &= \mathit{fun}(f{:}R{\rightarrow}\mathit{Object}(N))\ \ \mathit{fun}(s{:}R)\ \mathit{map}_{N}\ [R]\ [\mathit{Object}(N)]\ f\ (m\ s)\\ &:\ (R{\rightarrow}\mathit{Object}(N')) \rightarrow (R{\rightarrow}\mathit{Object}(N)).\end{aligned}$$

From this we obtain the desired result using EQ-FIX-SUB and EQ-APP.

For EQ-GM-SUB, the result follows directly from the definition using EQ-TBETA twice plus EQ-REFL.

EQ-OBJ-MAP follows from EQ-TBETA and EQ-BETA (several times), EQ-FIX, and EQ-ETA. □

7 F-bounded Quantification

If objects are modeled using recursive types, the higher-order quantification in the type of the generic method can be eliminated in favor of a specialized form of second-order quantification called *F-bounded quantification* (Canning *et al.*, 1989; Cook *et al.*, 1990), 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 (Abadi, 1992; Bruce, 1994):

$$\mathit{All}(A \le F(A))\ \ S \quad \approx \quad \mathit{All}(G \le F)\ \ [(\mu(A)G(A))/A]S.$$

Indeed, it follows from an observation by Abadi (1992) that, "in many models," these two types denote the same collection.

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

$$\mathit{GM}_N\ :\ \mathit{All}(A \le N(A))\ \ A \rightarrow N(A)$$

to match the expected types of programs manipulating objects. It is interesting to note that this also happens to be the simplest type of *classes* in the framework proposed by Cook, Canning, and Hill in (1990).

8 High-Level Syntax

One application of this abstract framework is that it yields a uniform syntax for compactly declaring object types and their associated message-sending operations. For example, the declaration

```
Point = ObjectType(Rep) with set: Int->Rep, get: Int
```

abbreviates the following set of declarations:

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

Similarly,

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

stands for

```
CPointN     = Fun(Rep) {set:Int->Rep, get:Int,
                        setC:Color->Rep, getC:Color}
CPoint'set  = fun(N<CPointN) fun(p:Object(N)) (GM_CPointN[N](p)).set
CPoint'get  = fun(N<CPointN) fun(p:Object(N)) (GM_CPointN[N](p)).get
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 in Section 5 (Pierce & Turner, 1994).

9 Mixed Variance

In this section, we consider extending our theory with mixed-variance signatures to account for binary methods. As we saw in Section 2, this cannot be accomplished simply by dropping the positivity requirement on N. We must deal with the fact that a binary method can be applied to objects with different representation types. Our running example will be points with equality, described by the mixed-variance signature:

```
EqPointN(X) = {set:Int->X, get:Int, eq:X->bool}
```

where the third method is used to compare two points for equality. Of course, in the framework developed so far we could write an external equality *function* (as opposed to an equality *method* local to one of the points) using the `get` method of both points generically:

```
eqPoint = fun(p:Point) fun(q:Point) eqInt (Point'get p) (Point'get q).
```

But it may be desirable to package the equality test with the rest of the methods of points. In this case, an implementation of the `eq` method must accept an arbitrary point as its second argument and interrogate it to obtain an integer coordinate. Our aim here is to give a formal account of this construction.

As always, we first state our requirements in an abstract form and try to give a suitable axiomatization. We then observe that mixed-variance objects directly imply possible non-termination of programs; thus no encoding is possible in a strongly normalizing type-theory like pure $F^{\omega}_{\leq}$. Finally, we show how recursive types may be used in combination with an arbitrary encoding of covariant objects to implement mixed-variance objects.

9.1 Mixed-Variance Objects

In general, let $N : \mathit{Type}{\rightarrow}\mathit{Type}$ be a mixed-variance type operator. We write N in the form $N = \mathit{Fun}(X)\ \ F\ X\ X$ for some binary operator F that is positive in its second argument. For example, the signature of points with equality is written

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

This technique of separating positive and mixed parts of an object signature forms the basis of the following definition of mixed-variance objects.

9.1.1 Definition: We write $\mathit{mixed}(F)$ to indicate that F is positive in its second argument:

$$\mathit{mixed}(F) \ \text{ iff } \ \mathit{pos}(F\ X),$$

where $F\ X = \mathit{Fun}(Y)\ \ FXY\ :\ \mathit{Type} \rightarrow \mathit{Type}$. Notice that for a given N there may be more than one such F; for example, we always have the trivial $F = \mathit{Fun}(X)\ \ \mathit{Fun}(Y)\ \ N(X)$. However, the type of the method implementation depends on the chosen F. In the trivial case, we have $FOR = NO$, independent of R, so that the methods must always return proper objects, not bare elements of the representation type. Such objects are completely degenerate, in the sense that the methods cannot modify the state. It is therefore better to push as much as possible of N into the positive part of F.

The well-formedness rule for mixed-variance object types and the typing of the associated generic method are straightforward generalizations of the ones in Section 3:

$$\frac{\Gamma \vdash F\ :\ \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type}}{\mathit{MObject}(F)\ :\ \mathit{Type}} \qquad \text{(K-MObj)}$$

$$\frac{\Gamma \vdash F\ :\ \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type} \qquad \mathit{mixed}(F)}{\Gamma \vdash \mathit{MGM}_F\ :\ \mathit{MObject}(F) \rightarrow (F\ \mathit{MObject}(F)\ \mathit{MObject}(F))} \qquad \text{(T-MGM*)}$$

For example:

```
MGM_EqPointF : EqPoint -> {set: Int->EqPoint,
                            get: Int,
                            eq: EqPoint->Bool},
```

where `EqPoint=MObject(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 a different representation type. Since, in a sense, $\mathit{MObject}(F)$ subsumes all other representation types, we can write the rule this way:

$$\frac{\begin{array}{c}\Gamma \vdash F\ :\ \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type} \qquad \mathit{mixed}(F) \\ \Gamma \vdash R\ :\ \mathit{Type} \qquad \Gamma \vdash s\ :\ R \qquad \Gamma \vdash m\ :\ R \rightarrow (F\ \mathit{MObject}(F)\ R)\end{array}}{\Gamma \vdash \mathit{mobject}_F\,(R,\, s,\, m)\ :\ \mathit{MObject}(F)} \qquad \text{(T-MObj-I)}$$

For example, point objects with equality can be created by

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

where

```
m = fun(s:Int)
     {set = fun(i:Int) i,
      get = s,
      eq = fun(p:EqPoint) eqInt s (MGM_EqPointF p).get}
```

and tested for equality in expressions like

```
EqPoint'eq (mkeqpoint 5) (mkeqpoint 6),
```

where `EqPoint'eq` is defined by projection from the generic method as in Section 8.

As in the specification of GM in Section 3, the observation that methods can only manipulate other objects generically underlies the equational specification of MGM. If R is some representation type and $m : R \to (F\ MObject(F)\ R)$ is an implementation of the methods according to the above rule, then EQ-OBJ-MAP* generalizes to

$$\begin{tikzcd} O \arrow[r, "MGM_F"] & FOO \\ R \arrow[u, "mobject_F\,(R,\,—,\,m)"] \arrow[r, "m"'] & FOR \arrow[u, "map_{(FO)}\ [R]\ [O]\ mobject_F\,(R,\,—,\,m)"'] \end{tikzcd} \qquad \text{(EQ-MOBJ-MAP*)}$$

where $O = MObject(F)$ and, as before, $FO = Fun(Y)\,FOY$.

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

```
(MGM_EqPointF (mkpoint x)).eq y  =  eqInt (MGM_EqPointF y).get x.
```

This again reflects the intuition that the generic method should not perform any computation of its own, but should simply invoke the local implementation.

9.2 Mixed-Variance Subtyping

The typing of the generic method can now be extended to handle subtyping exactly as in Section 4:

$$\frac{\Gamma \vdash F \ :\ Type \to Type \to Type \qquad mixed(F)}{\Gamma \vdash MGM_F \ :\ All(F' \leq F)\ \ MObject(F') \to (F\ \ MObject(F')\ \ MObject(F'))} \qquad \text{(T-MGM)}$$

Following the development in Section 4, we generalize the diagram specifying the behavior of the generic method to make it account for the more refined type:

$$\begin{tikzcd} O \arrow[r, "MGM_F\ [F]"] & FOO \\ R \arrow[u, "mobject_F\,(R,\,—,\,m)"] \arrow[r, "m"'] & FOR \arrow[u, "map_{(FO)}\ [R]\ [O]\ mobject_F\,(R,\,—,\,m)"'] \end{tikzcd} \qquad \text{(EQ-MOBJ-MAP)}$$

where again $O = MObject(F)$.

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

```
MGM_EqPointF : All(F'<EqPointF)
                 MObject(F') -> {set: Int->MObject(F'),
                                 get: Int,
                                 eq: MObject(F')->Bool}.
```

Subtyping between object types must be defined by a more restrictive rule than the one we used in Section 4:

$$\frac{\Gamma \vdash F' \sqsubseteq F}{\Gamma \vdash MObject(F') \leq MObject(F)} \qquad \text{(S-MObj)}$$

where

$$\Gamma \vdash F' \sqsubseteq F \quad \text{iff} \quad \Gamma,\, B_1, B_2 {\leq} Top(Type),\, A_1{\leq}B_1,\, A_2{\leq}B_2 \vdash F'A_1A_2 \leq FB_1B_2.$$

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) Fun(Y) {set: Int->Y, get: Int, eq: X->bool,
                           setC: Color->Y, getC: Color}.
```

If we allowed `MObject(EqCPointF)` $\leq$ `MObject(EqPointF)`, then we could break the type system by creating an instance `cp` of `MObject(EqCPointF)` whose equality method called the `getC` method of its argument, promoting `cp` to type `MObject(EqPointF)`, and invoking its `eq` method with an instance of `MObject(EqPointF)` as argument (c.f. Cook *et al* (1990) and Bruce (1994)). This observation might lead one to wonder whether there are *any* nontrivial subtyping relations between mixed-variance object types. Indeed, it seems that $F' \sqsubseteq F$ cannot hold when both F' and F are non-constant in their first arguments. However, note that we do have, for example, `EqPointF` $\sqsubseteq$ `PointF = Fun(X) Fun(Y) PointN(Y)`.

The rules Eq-Obj-Sub and Eq-GM-Sub are extended as follows:

$$\frac{\begin{array}{c}\Gamma \vdash F' \sqsubseteq F \qquad \Gamma \vdash F'' \leq F' \qquad \Gamma \vdash F'' \leq F \\ mixed(F) \qquad mixed(F')\end{array}}{\begin{array}{l}\Gamma \vdash MGM_{F'}\ [F''] = MGM_F\ [F''] \\ \quad :\ MObject(F'') \rightarrow (F\ MObject(F'')\ MObject(F''))\end{array}} \qquad \text{(Eq-MGM-Sub)}$$

$$\frac{\begin{array}{c}\Gamma \vdash F' \sqsubseteq F \qquad \Gamma \vdash s\ :\ R \\ \Gamma \vdash m\ :\ R \rightarrow (F'\ MObject(F')\ R) \\ mixed(F) \qquad mixed(F')\end{array}}{\Gamma \vdash mobject_{F'}\,(R,\, s,\, m) = mobject_F\,(R,\, s,\, m)\ :\ MObject(F)} \qquad \text{(Eq-MObj-Sub)}$$

9.3 Mixed Variance Implies Non-Termination

It may surprise the reader to learn that an implementation of mixed-variance objects satisfying the specification Eq-MObj-Map can be used to solve fixed-point equations on terms, even in the absence of explicit type- or value-level recursion in the ambient type theory.

Let $\phi : D \rightarrow D$ be a function whose fixed point we wish to calculate, and let

$$\begin{array}{lcl} F\,X\,Y & = & X \rightarrow D \\ O & = & MObject(F) \\ g & = & MGM_F\,[F] \\ & : & MObject(F) \rightarrow MObject(F) \rightarrow D. \end{array}$$

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

$$\begin{array}{lcl} m & = & fun(x{:}Top)\ fun(y{:}O)\ \phi(g\ y\ y) \\ & : & Top \rightarrow O \rightarrow D \\ & = & Top \rightarrow F\ O\ Top \end{array}$$

as the concrete method

$$\begin{array}{lcl} o & = & mobject_F\,(Top,\ top,\ m) \\ & : & O, \end{array}$$

where top is any element of Top. It follows from EQ-MOBJ-MAP that $(g\ o\ o)$ is a fixed point of ϕ. To see this, observe that F is constant in its positive argument, so, by the second clause of Definition B.6, $map_{(F\,O)}\ [U]\ [V]\ f$ is the identity function on $O \rightarrow D$ no matter what $f : U \rightarrow V$ is. So we can calculate as follows:

$$\begin{array}{ll} & g\ o\ o \\ = & g\ (mobject_F\,(Top,\ top,\ m))\ o \\ = & id\,[O \rightarrow D]\ (m\ top\ o) \\ = & \phi\ (g\ o\ o). \end{array}$$

Thus, we cannot hope to find an implementation of mixed-variance signatures in a strongly normalizing system like F^ω with existential types.

It might be the case that an implementation could be given in F^ω augmented with general recursion at the level of terms, without introducing recursive types. Indeed, in an earlier presentation of this work (Hofmann & Pierce, 1992), we gave an encoding of a slightly different version of mixed-variance signatures in F^ω extended with a fixed-point operator at the level of values. Mixed-variance objects were encoded as existential packages, with mixed-variance methods abstracted over polymorphic functions representing the generic method. However, we found this solution contrived; moreover, it could not satisfactorily be extended with subtyping.

9.4 Mixed Variance Objects via Recursive Types

On the other hand, a natural implementation of mixed-variance objects can easily be given in terms of recursive types. Suppose we are given an implementation of the covariant object constructors *Object*, *GM*, and *object* that satisfies the requirements set out in the previous sections. Now, let

$$MObject(F) = \mu(X)\ Object(FX).$$

Recall that $mixed(F)$ means just $pos(F\ X)$. Next, let

$$\begin{array}{lcl} MGM_F & = & fun(F' {\leq} F)\ \ GM_{F\ MObject(F')}\ [F'\ MObject(F')] \\ & : & All(F' {\leq} F)\ \ Object(F'\ MObject(F')) \\ & & \qquad\qquad \rightarrow F\ MObject(F')\ Object(F'\ MObject(F')) \\ & \sim & All(F' {\leq} F)\ \ MObject(F') \rightarrow (F\ MObject(F')\ MObject(F')). \end{array}$$

Note that the bounded quantifier in the type of the generic method refers to pointwise subtyping of type operators, as always — not to monotone subtyping. Finally, if $R : Type$ and $s : R$ and $m : R \rightarrow (F\ MObject(F)\ R)$, then let

$$\begin{array}{lcl} mobject_F\,(R,\,s,\,m) & = & object_{F\ MObject(F)}\,(R,\,s,\,m) \\ & : & Object(F\ MObject(F)) \\ & \sim & MObject(F). \end{array}$$

The required laws follow from the laws governing covariant objects.

If we instantiate the covariant object constructors with the concrete implementation in terms of recursive types from Section 6, we obtain

$$MObject(F) = \mu(X)\ \mu(Y)\ F\ X\ Y,$$

which denotes the same regular tree as the familiar encoding

$$MObject(F) = \mu(X)\ F\ X\ X$$

proposed, for example, in (Cardelli, 1984; Cardelli, 1992; Bruce, 1994; Mitchell, 1990a; Canning *et al.*, 1989; Cook *et al.*, 1990). On the other hand, the existential encoding of objects of Section 5 leads to the implementation

$$MObject(F) = \mu(X)\ Some(A)\ \ \{state : A,\ methods : A \rightarrow F\ X\ A\}$$

of mixed-variance objects, as suggested by one of the referees.

10 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 previously studied encodings of objects to be presented in a common setting.

The object encodings using existential types and recursive types essentially coincide for positive method signatures. Extending these encodings to mixed-variance signatures, on the other hand, seems to require recursive types. However, it might be argued that 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 non-termination 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 the ones that are most often needed in practice, since they can only access the concrete representation of one of their arguments. It may be 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. This perspective can be fully integrated with object-style programming, as shown by Pierce and Turner (1993).

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. Cook *et al* (1990)). It can be shown that, once the fundamental mechanisms of encapsulation and subtyping are accounted for and their interaction properly handled, inheritance, including features like `self` and `super`, arises as a collection of programming idioms completely within the resulting type theory (Pierce & Turner, 1994). It would thus be a straightforward matter to extend the abstract framework developed here to include an implementation of inheritance, using the ideas developed in (Cook, 1989; Kamin & Reddy, 1994; Cook *et al.*, 1990; Bruce, 1994; Cardelli, 1992; Mitchell, 1990a; Pierce & Turner, 1994).

Another application of our framework may lie in suggesting appropriate proof rules for the verification of object-oriented programs. Here, 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. Some preliminary results in this direction are reported in (Hofmann & Pierce, 1994).

Acknowledgements

David N. Turner joined in many discussions and helped formalize the definitions of positivity and *map* in Section 5. Marcelo Fiore pointed out a relationship between terminal algebras and recursive types that led to the axiomatization of mixed-variance methods in Section 9. 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 supplied pointers to relevant literature. Terry Stroup and Zdzisław Spławski gave us helpful suggestions on earlier drafts. Two anonymous referees made numerous suggestions, which led to substantial reworking and improvement of both technical aspects and exposition.

This research was mainly carried out at the University of Edinburgh's Lab for Foundations of Computer Science. Hofmann was supported by a European Union HCM fellowship. Pierce was supported by a fellowship from the British Science and Engineering Research Council. Earlier versions of this paper appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

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} (1972) with subtyping. The organizing ideas behind the system are due to Cardelli, particularly to the 1988 paper, "Structural Subtyping and the Notion of Power Type" (1988); the extension of the subtype relation to type operators was invented by Cardelli and Mitchell (Cardelli, 1990; Mitchell, 1990a; Bruce & Mitchell, 1992; Abadi, 1992). Cardelli's sketch (1990) also suggests a more powerful treatment of operator subtyping, including both monotonic and antimonotonic subtyping in addition to pointwise subtyping; but the metatheoretic properties of this extension are not well understood.

The metatheory of pure $F^{\omega}_{\leq}$ has been studied by Steffen and Pierce (1994) and Compagnoni (1994). Simple models have been given by Cardelli and Longo (1991) and Compagnoni and Pierce (1993). A model for an extension of $F^{\omega}_{\leq}$ with recursive types has been given by Bruce and Mitchell (1992).

A.1 Syntax

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

K	$::=$	$Type$	kind of types
	$\mid$	$K_1 \rightarrow K_2$	kind of type operators
T	$::=$	A	type variable
	$\mid$	$Fun\,(A{:}K)\,T$	type operator
	$\mid$	$T_1\ T_2$	application of a type operator
	$\mid$	$Top(K)$	maximal type of kind K
	$\mid$	$T_1 \rightarrow T_2$	function type
	$\mid$	$All(A{\leq}T_1)\,T_2$	universally quantified type
	$\mid$	$\{l_1{:}T_1,\ \ldots,\ l_n{:}T_n\}$	record type
e	$::=$	x	variable
	$\mid$	$fun\,(x{:}T)\,e$	abstraction
	$\mid$	$e_1\ e_2$	application
	$\mid$	$fun\,(A{\leq}T)\,e$	type abstraction
	$\mid$	$e\ [T]$	type application
	$\mid$	$\{l_1 = e_1,\ \ldots,\ l_n = e_n\}$	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:

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

We sometimes write $\Gamma \vdash S \sim T$ to mean that both $\Gamma \vdash S \leq T$ and $\Gamma \vdash T \leq S$.

A.1.3 Definition: The *domain* of a context Γ, written $\mathrm{dom}(\Gamma)$, is the set of type and term variables bound by Γ. 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. $\Gamma \equiv \bullet$, 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 $\Gamma \vdash e : T$ 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 and allows a context to be viewed as a partial function, justifying the notation $\Gamma(X)$ for the unique upper bound of $X : \mathrm{dom}(\Gamma)$.

A.2 Conversion on Types

The $\beta\top$-conversion relation on type expressions is the least congruence (with respect to all of the type formers) containing the following rules:

$$\begin{array}{lcl} (\mathit{Fun}(A{:}K)\,T)\ S & =_{\beta\top} & [S/A]T \\ \mathit{Top}(K_1 \rightarrow K_2)\ S & =_{\beta\top} & \mathit{Top}(K_2). \end{array}$$

The corresponding reduction relation is defined in the usual way, orienting these rules from left to right. When T has a $\beta\top$-normal form, we write it as $T^{!}$.

A.3 Contexts

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

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

$$\frac{\Gamma \vdash T : \mathit{Type}}{\vdash \Gamma, x{:}T \text{ context}} \qquad \text{(C-VAR)}$$

A.4 Kinding

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

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

$$\frac{\Gamma \vdash S \ : \ K_1{\rightarrow}K_2 \qquad \Gamma \vdash T \ : \ K_1}{\Gamma \vdash S\ T \ : \ K_2} \qquad \text{(K-App)}$$

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

$$\frac{\Gamma \vdash T_1 \ : \ \mathit{Type} \qquad \Gamma \vdash T_2 \ : \ \mathit{Type}}{\Gamma \vdash T_1{\rightarrow}T_2 \ : \ \mathit{Type}} \qquad \text{(K-Arrow)}$$

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

$$\frac{\vdash \Gamma \ \text{context} \qquad \text{for each } i,\ \ \Gamma \vdash T_i \ : \ \mathit{Type}}{\Gamma \vdash \{l_1{:}T_1,\, \ldots,\, l_n{:}T_n\} \ : \ \mathit{Type}} \qquad \text{(K-Rcd)}$$

A.5 Subtyping

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

$$\Gamma \vdash A \leq \Gamma(A) \qquad \text{(S-TVar)}$$

$$\Gamma \vdash T \leq T \qquad \text{(S-Refl)}$$

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

$$\frac{\Gamma \vdash S \ : \ K}{\Gamma \vdash S \leq \mathit{Top}(K)} \qquad \text{(S-Top)}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1{\rightarrow}S_2 \leq T_1{\rightarrow}T_2} \qquad \text{(S-Arrow)}$$

$$\frac{\Gamma,\, A{\leq}U \vdash S_2 \leq T_2}{\Gamma \vdash \mathit{All}(A{\leq}U)\,S_2 \leq \mathit{All}(A{\leq}U)\,T_2} \qquad \text{(S-All)}$$

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

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

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

A.6 Typing

$$\frac{\Gamma \vdash e \;:\; S \qquad \Gamma \vdash S \leq T \qquad \Gamma \vdash T \;:\; K}{\Gamma \vdash e \;:\; T} \qquad \text{(T-Subsumption)}$$

$$\frac{\vdash \Gamma \;\text{context}}{\Gamma \vdash x \;:\; \Gamma(x)} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\, x{:}T_1 \vdash e \;:\; T_2}{\Gamma \vdash \mathit{fun}\,(x{:}T_1)\,e \;:\; T_1 {\rightarrow} T_2} \qquad \text{(T-Arrow-I)}$$

$$\frac{\Gamma \vdash f \;:\; T_1 {\rightarrow} T_2 \qquad \Gamma \vdash a \;:\; T_1}{\Gamma \vdash f\; a \;:\; T_2} \qquad \text{(T-Arrow-E)}$$

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

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

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

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

A.7 Basic Properties

Proofs of the following can be found in Steffen and Pierce (1994). Strictly speaking, the development there does not explicitly deal with record types, but this is a straightforward extension: proof-theoretically, the record constructor behaves just like other simple constructors such as arrow.

A.7.1 Proposition [Strong normalization of well-kinded types]: If $\Gamma \vdash T \;:\; K$, then T has a unique $\beta\top$-normal form.

A.7.2 Definition [Promotion]: The *promotion* of a type $A\; S_1 \ldots S_n$ in a well-formed context Γ is $\Gamma(A)\; S_1 \ldots S_n$. We write $A\; S_1 \ldots S_n \uparrow_\Gamma \Gamma(A)\; S_1 \ldots S_n$.

A.7.3 Proposition: The following algorithm is sound and complete for the relation $\Gamma \vdash S \leq T$, when S and T are well-kinded.

$\mathit{check}(\Gamma \vdash S \leq T) =$
  $\mathit{check}'(\Gamma \vdash S' \leq T')$

$\mathit{check}'(\Gamma \vdash S \leq T) =$
  if $T = \mathit{Top}(\mathit{Kind}_\Gamma(S))$
    then true
  else if $S = T$
    then true
  else if $S \uparrow_\Gamma U$
    then $\mathit{check}'(\Gamma \vdash U' \leq T)$
  else if $S = S_1 {\rightarrow} S_2$ *and* $T = T_1 {\rightarrow} T_2$
    then $\quad \mathit{check}'(\Gamma \vdash T_1 \leq S_1)$

and $check^!(\Gamma \vdash S_2 \leq T_2)$
else if $S = All(A{\leq}U)\, S_2$ *and* $T = All(A{\leq}U)\, T_2$
then $check^!(\Gamma,\, A{\leq}U \vdash S_2 \leq T_2)$
else if $S = \{k_1{:}S_1, \ldots, k_m{:}S_m\}$ *and* $T = \{l_1{:}T_1, \ldots, l_n{:}T_n\}$
then $\{l_1, \ldots, l_n\} \subseteq \{k_1, \ldots, k_m\}$
and for each $k_i = l_j$, $check^!(\Gamma \vdash S_i \leq T_j)$
else if $S = Fun(A{:}K_1)\, S_2$ *and* $T = Fun(A{:}K_1)\, T_2$
then $check^!(\Gamma,\, A{\leq}\, Top(K_1) \vdash S_2 \leq T_2)$
else
false.

A.8 Equational Theory

Developing a full-fledged equational theory for $F^{\omega}_{\leq}$ remains a matter for future research. For the sake of concreteness, we propose the following rules, which straightforwardly generalize the equational theory for pure second-order quantification studied by Cardelli, Martini, Mitchell, and Scedrov (1994). All of the rules are sound in a standard PER model of $F^{\omega}_{\leq}$ (see, for example, Compagnoni and Pierce (1993)). We do not aim for a minimal set of rules. To reduce clutter, we elide the evident well-kindedness premises; these can be filled in by analogy with the typing rules in the previous section.

$$\frac{\Gamma \vdash e \;:\; T}{\Gamma \vdash e = e \;:\; T} \qquad \text{(Eq-Refl)}$$

$$\frac{\Gamma \vdash e = e' \;:\; T}{\Gamma \vdash e' = e \;:\; T} \qquad \text{(Eq-Symm)}$$

$$\frac{\Gamma \vdash e = e' \;:\; T \qquad \Gamma \vdash e' = e'' \;:\; T}{\Gamma \vdash e = e'' \;:\; T} \qquad \text{(Eq-Trans)}$$

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

$$\frac{\Gamma \vdash e \;:\; Top(Type) \qquad \Gamma \vdash e' \;:\; Top(Type)}{\Gamma \vdash e = e' \;:\; Top(Type)} \qquad \text{(Eq-Top)}$$

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

$$\frac{\Gamma \vdash f = f' \;:\; S {\rightarrow} T \qquad \Gamma \vdash a = a' \;:\; S}{\Gamma \vdash f\ a = f'\ a' \;:\; T} \qquad \text{(Eq-App)}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\leq}U \vdash T \leq T' \\ \Gamma,\, A{\leq}U \vdash b = b' \;:\; T\end{array}}{\Gamma \vdash fun\,(A{\leq}U)\, b = fun\,(A{\leq}U)\, b' \;:\; All(A{\leq}U)\, T'} \qquad \text{(Eq-TAbs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e = e' \;:\; All(A{\leq}U)\, V \\ \Gamma \vdash S \leq U \qquad \Gamma \vdash S' \leq U \\ \Gamma \vdash [S/A]V \leq T \qquad \Gamma \vdash [S'/A]V \leq T\end{array}}{\Gamma \vdash e\ [S] = e'\ [S'] \;:\; T} \qquad \text{(Eq-TApp)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash e_i = e_i' \ :\ T_i}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} = \{l_1 = e_1', \ldots, l_n = e_n'\} \ :\ \{l_1{:}T_1, \ldots, l_n{:}T_n\}} \quad (\textsc{Eq-Rcd})$$

$$\frac{\Gamma, x{:}S \vdash b = b' \ :\ T \qquad \Gamma \vdash a = a' \ :\ S}{\Gamma \vdash (\mathit{fun}\,(x{:}S)\, b)\ a = [a'/x]b' \ :\ T} \quad (\textsc{Eq-Beta})$$

$$\frac{\Gamma \vdash f = f' \ :\ S \rightarrow T}{\Gamma \vdash \mathit{fun}\,(x{:}S)\ f\ x = f' \ :\ S \rightarrow T} \quad (\textsc{Eq-Eta})$$

$$\frac{\Gamma, A{\leq}S \vdash b = b' \ :\ T \qquad \Gamma \vdash U \leq S}{\Gamma \vdash (\mathit{fun}\,(A{\leq}S)\, b)\ [U] = [U/A]b' \ :\ [U/A]T} \quad (\textsc{Eq-TBeta})$$

$$\frac{\Gamma \vdash f = f' \ :\ \mathit{All}(A{\leq}S)\, T}{\Gamma \vdash \mathit{fun}\,(A{\leq}S)\ f\ [A] = f' \ :\ \mathit{All}(A{\leq}S)\, T} \quad (\textsc{Eq-TEta})$$

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

$$\frac{\Gamma \vdash r \ :\ \{l_1{:}T_1, \ldots, l_n{:}T_n\}}{\Gamma \vdash r = \{l_1 = r.\ l_1, \ldots, l_n = r.\ l_n\} \ :\ \{l_1{:}T_1, \ldots, l_n{:}T_n\}} \quad (\textsc{Eq-Surj})$$

A.8.1 Remark: It might appear that the rule Eq-TAbs should be generalized, by analogy with Eq-Abs, to allow the comparison of type abstractions with different upper bounds. Similarly, one might wish to generalize the rules Eq-Map-Sub and Eq-GM-Sub in Appendix B to allow the direct comparison of two different mapping functions or generic methods, instead of comparing particular instances. But for such equations even to typecheck, it would first be necessary to similarly generalize the subtyping rule S-All, leading to a richer system, but one with a much more difficult metatheory. See Steffen and Pierce (1994) for a related discussion.

The rule Eq-TApp is closely related to the semantic concept of parametricity (c.f. (Cardelli *et al.*, 1994)).

A.8.2 Fact: The following more general version of Eq-Abs is derivable using Eq-Abs and transitivity:

$$\frac{\Gamma \vdash S' \leq S \qquad \Gamma \vdash T \leq T' \qquad \Gamma, x{:}S' \vdash b' = b \ :\ T \qquad \Gamma, x{:}S \vdash b \ :\ T}{\Gamma \vdash \mathit{fun}\,(x{:}S')\, b' = \mathit{fun}\,(x{:}S)\, b \ :\ S' \rightarrow T'} \quad (\textsc{Eq-Abs}^{+})$$

B Positivity

In the body of the article, we stipulated that the "ambient type theory" should come equipped with a positivity predicate *pos* and an operator *map* satisfying the following laws:

$$\frac{\Gamma \vdash N \ :\ \mathit{Type} \rightarrow \mathit{Type} \qquad \mathit{pos}(N) \qquad \Gamma \vdash S \leq T}{\Gamma \vdash N\, S \leq N\, T} \quad (\textsc{S-Pos-Mono})$$

$$\frac{\Gamma \vdash N \ :\ \mathit{Type} \rightarrow \mathit{Type} \qquad \mathit{pos}(N)}{\Gamma \vdash \mathit{map}_N \ :\ \mathit{All}(X)\, \mathit{All}(Y)\ (X \rightarrow Y) \rightarrow (N(X) \rightarrow N(Y))} \quad (\textsc{T-Map})$$

$$\frac{\begin{array}{c}\Gamma \vdash N \;:\; \mathit{Type}{\rightarrow}\mathit{Type} \qquad \mathit{pos}(N) \\ \Gamma \vdash f \;:\; X{\rightarrow}Y \qquad \Gamma \vdash g \;:\; Y{\rightarrow}Z\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{map}_N\;[X]\;[Z]\;(f;g) \\ \quad = (\mathit{map}_N\;[X]\;[Y]\;f)\;;\;(\mathit{map}_N\;[Y]\;[Z]\;g) \\ \quad :\; N(X){\rightarrow}N(Z)\end{array}} \qquad \text{(EQ-MAP-TRANS)}$$

$$\frac{\Gamma \vdash N \;:\; \mathit{Type}{\rightarrow}\mathit{Type} \qquad \mathit{pos}(N) \qquad \Gamma \vdash X \;:\; \mathit{Type}}{\Gamma \vdash \mathit{map}_N\;[X]\;[X]\;(\mathit{id}\,[X]) = \mathit{id}\,[N(X)] \;:\; N(X){\rightarrow}N(X)} \qquad \text{(EQ-MAP-ID)}$$

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \qquad \mathit{pos}(N') \qquad \mathit{pos}(N) \\ \Gamma \vdash f \;:\; X{\rightarrow}Y \qquad \Gamma \vdash n \;:\; N'(X)\end{array}}{\Gamma \vdash \mathit{map}_{N'}\;[X]\;[Y]\;f\;n = \mathit{map}_N\;[X]\;[Y]\;f\;n \;:\; N(Y)} \qquad \text{(EQ-MAP-SUB)}$$

In this section, we show how such a *pos* and *map* can be defined for $F^{\omega}_{\leq}$.

B.1 Remark: Of course, the requirement that a given type theory should include positivity can always be satisfied trivially by setting $\mathit{pos}(N) = \mathit{false}$ for all N. Furthermore, as we remark below, the following definitions of *pos* and *map* — which essentially follow the prevailing type-theoretic "folklore" — are less complete than one might wish, in the sense that they do not handle arbitrary operators containing higher-order variables. However, they are strong enough to establish the positivity of many reasonable object interfaces.

B.2 Definition: Let T be a type that is closed in some context where a type variable A is defined. Define the predicates $\mathit{pos}_A(T)$ ("A occurs only positively in T") and $\mathit{neg}_A(T)$ ("A occurs only negatively in T") simultaneously as follows:

$$\begin{array}{lcl}
\mathit{pos}_A(A) & = & \text{true} \\
\mathit{pos}_A(T) \;\text{ with } A \text{ not free in } T & = & \text{true} \\
\mathit{pos}_A(T_1{\rightarrow}T_2) & = & \mathit{neg}_A(T_1) \text{ and } \mathit{pos}_A(T_2) \\
\mathit{pos}_A(\mathit{All}(B{\leq}T_1)\,T_2) & = & A \text{ not free in } T_1 \text{ and } \mathit{pos}_A(T_2) \\
\mathit{pos}_A(\{l_1{:}T_1,\;\ldots,\;l_n{:}T_n\}) & = & \mathit{pos}_A(T_i) \text{ for all } i \\
\mathit{pos}_A(T) \text{ in all other cases} & = & \text{false} \\
 & & \\
\mathit{neg}_A(A) & = & \text{false} \\
\mathit{neg}_A(T) \text{ with } A \text{ not free in } T & = & \text{true} \\
\mathit{neg}_A(T_1{\rightarrow}T_2) & = & \mathit{pos}_A(T_1) \text{ and } \mathit{neg}_A(T_2) \\
\mathit{neg}_A(\mathit{All}(B{\leq}T_1)\,T_2) & = & A \text{ not free in } T_1 \text{ and } \mathit{neg}_A(T_2) \\
\mathit{neg}_A(\{l_1{:}T_1,\;\ldots,\;l_n{:}T_n\}) & = & \mathit{neg}_A(T_i) \text{ for all } i \\
\mathit{neg}_A(T) \text{ in all other cases} & = & \text{false}
\end{array}$$

B.3 Definition: If $\Gamma \vdash N \;:\; \mathit{Type}{\rightarrow}\mathit{Type}$, we write $\mathit{pos}(N)$ to mean that $N^{!}$, the normal form of N, equals $\mathit{Fun}(A)\,P$ and $\mathit{pos}_A(P)$.

Notice, in particular, that $\mathit{pos}(N)$ is always false when N is a variable. Similarly, an application $N(S)$ can only be marked pos_A or neg_A if A does not occur free in N or S. As we remarked in Section 3, a stronger calculus (c.f. (Cardelli, 1990)) with positive and negative (monotone and antimonotone) operators seems cleaner in this respect, since it allows positivity/negativity to be ascribed to more type expressions. However, the present formulation is sufficient for our purposes.

An occurrence of A in the bound of a quantifier is never considered to be only positive or only negative: $pos_A(All(B{\leq}T_1)\,T_2)$ and $neg_A(All(B{\leq}T_1)\,T_2)$ 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 *map* 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 situation where A occurs in T_1.) The same observation will apply to the bounds of existential quantifiers.

It is easy to check that every positive operator is also monotone in the subtype relation, validating S-Pos-Mono:

B.4 Lemma: Let $\Gamma = \Delta_1,\, A{\leq}\,Top(Type),\, \Delta_2$ and $\Delta = \Delta_1, \Delta_2$, with A not free in Δ_2, and suppose that $\Gamma \vdash P \;:\; Type$ and $\Delta \vdash S \leq T$.

1. If $pos_A(P)$, then $\Delta \vdash [S/A]P \leq [T/A]P$.
2. If $neg_A(P)$, then $\Delta \vdash [T/A]P \leq [S/A]P$.

Proof: By simultaneous induction on the definitions of pos_A and neg_A. For each part, the first two cases are easy, the next three follow by straightforward use of the induction hypothesis, and the last one holds trivially. □

B.5 Corollary: [Positivity implies monotonicity]

1. If $\Gamma \vdash N \;:\; Type{\rightarrow}Type$ and $pos(N)$, then $\Gamma \vdash S \leq T$ implies $\Gamma \vdash N\,S \leq N\,T$.
2. If $\Gamma \vdash N \;:\; Type{\rightarrow}Type$ and $neg(N)$, then $\Gamma \vdash S \leq T$ implies $\Gamma \vdash N\,T \leq N\,S$.

B.6 Definition: Fix a context Γ, a type variable A, two types X and Y, and a function $f \;:\; X{\rightarrow}Y$. Then the *lifting* of f through a type T, written $lift_T^{A\leftarrow f}$, is

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

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

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

Note that $lift_T^{A\leftarrow f}$ is defined iff $pos_A(T)$, that $lower_T^{A\leftarrow f}$ is defined iff $neg_A(T)$, and that

$$
\begin{array}{lll}
lift_T^{A\leftarrow f} & : & [X/A]T \rightarrow [Y/A]T \\
lower_T^{A\leftarrow f} & : & [Y/A]T \rightarrow [X/A]T.
\end{array}
$$

Intuitively, $lift_T^{A\leftarrow f}$ "maps f over elements of T" by destructing and rebuilding an element of T, inserting a call to f at each occurrence of the variable A (all of which are positive); $lower_T^{A\leftarrow f}$ does the same for negative occurrences of A. The change in sign of the occurrences of A accounts for the switch of X and Y in the types of $lift_T^{A\leftarrow f}$ and $lower_T^{A\leftarrow f}$. More abstractly, $lift_T^{A\leftarrow f}$ witnesses the fact that every positive type operator induces a functor — i.e. a pair of maps, one on types and one on terms.

B.7 Lemma: Suppose A is not free in Δ. Then the following equality rules for *lift* and *lower* are derivable:

$$
\frac{\Gamma,\, A\leq Top(Type),\, \Delta \vdash T\ :\ Type \qquad pos_A(T) \qquad \Gamma \vdash X\ :\ Type}{\Gamma,\, \Delta \vdash lift_T^{A\leftarrow id[X]} = id\ [[X/A]T]\ :\ [X/A]T\rightarrow[X/A]T} \qquad \text{(Eq-Lift-Id)}
$$

$$
\frac{\Gamma,\, A\leq Top(Type),\, \Delta \vdash T\ :\ Type \qquad neg_A(T) \qquad \Gamma \vdash X\ :\ Type}{\Gamma,\, \Delta \vdash lower_T^{A\leftarrow id[X]} = id\ [[X/A]T]\ :\ [X/A]T\rightarrow[X/A]T} \qquad \text{(Eq-Lower-Id)}
$$

$$
\frac{\begin{array}{c}\Gamma,\, A\leq Top(Type),\, \Delta \vdash T\ :\ Type \qquad pos_A(T) \\ \Gamma \vdash f\ :\ X\rightarrow Y \qquad \Gamma \vdash g\ :\ Y\rightarrow Z\end{array}}{\begin{array}{l}\Gamma,\, \Delta \vdash lift_T^{A\leftarrow (f;g)} = (lift_T^{A\leftarrow f}\ ;\ lift_T^{A\leftarrow g}) \\ \quad :\ [X/A]T\rightarrow[Z/A]T\end{array}} \qquad \text{(Eq-Lift-Trans)}
$$

$$
\frac{\begin{array}{c}\Gamma,\, A\leq Top(Type),\, \Delta \vdash T\ :\ Type \qquad neg_A(T) \\ \Gamma \vdash f\ :\ X\rightarrow Y \qquad \Gamma \vdash g\ :\ Y\rightarrow Z\end{array}}{\begin{array}{l}\Gamma,\, \Delta \vdash lower_T^{A\leftarrow (f;g)} = (lower_T^{A\leftarrow g}\ ;\ lower_T^{A\leftarrow f}) \\ \quad :\ [Z/A]T\rightarrow[X/A]\end{array}} \qquad \text{(Eq-Lower-Trans)}
$$

Proof: Each pair of rules is proved simultaneously, by induction on the structure of T. The -ID pair uses EQ-ETA, EQ-TETA, and EQ-SURJ for the three inductive cases. The -TRANS pair uses EQ-BETA, EQ-TBETA, and EQ-PROJ. □

B.8 Lemma: Suppose A is not free in Δ. If T and T' are normal forms of kind *Type* in $\Gamma, A{\leq}Top(Type), \Delta$, then the following rules are derivable:

$$\frac{\begin{array}{c}\Gamma, A{\leq}Top(Type), \Delta \vdash T' \leq T \\ pos_A(T') \qquad pos_A(T) \qquad \Gamma \vdash f : X{\rightarrow}Y\end{array}}{\Gamma, \Delta \vdash lift_{T'}^{A\leftarrow f} = lift_T^{A\leftarrow f} \;:\; [X/A]T' \rightarrow [Y/A]T} \qquad \text{(EQ-LIFT-SUB)}$$

$$\frac{\begin{array}{c}\Gamma, A{\leq}Top(Type), \Delta \vdash T' \leq T \\ neg_A(T') \qquad neg_A(T) \qquad \Gamma \vdash f : X{\rightarrow}Y\end{array}}{\Gamma, \Delta \vdash lower_{T'}^{A\leftarrow f} = lower_T^{A\leftarrow f} \;:\; [Y/A]T \rightarrow [X/A]T'} \qquad \text{(EQ-LOWER-SUB)}$$

Proof: Both statements are proved by simultaneous induction on a successful execution of the algorithm $check^!$ applied to $\Gamma, A{\leq}Top(Type) \vdash T' \leq T$. We give the argument just for EQ-LIFT-SUB; the other is symmetric.

When $T = Top(Type)$, use EQ-TOP, EQ-ETA, and EQ-ABS. When $T = T'$, use EQ-REFL.

If $T' \Uparrow_\Gamma U$, then by the definition of promotion, T' has the form $B\ U_1\ \ldots\ U_n$. There are two cases to consider. If $B = A$, then since $A : Type$, we have $n = 0$, $T' = A$, and $U = Top(Type)$. If $B \neq A$, then A is not free in T', since $pos_A(T')$ implies that A does not occur in any of the U_i; since A is not free in Γ or Δ, it therefore cannot be free in U. Thus, we see that $pos_A(U)$ and $lift_{T'}^{A\leftarrow f}$ and $lift_U^{A\leftarrow f}$ are the same identity function. In either case, we have

$$\Gamma, \Delta \vdash lift_{T'}^{A\leftarrow f} = lift_U^{A\leftarrow f} \;:\; [X/A]T' \rightarrow U$$

(since A is not free in U). By the induction hypothesis, $\Gamma, \Delta \vdash lift_U^{A\leftarrow f} = lift_T^{A\leftarrow f} : U \rightarrow [Y/A]T$. The result follows by transitivity, using

$$\begin{array}{l}\Gamma, \Delta \vdash [X/A]T' \leq U \\ \Gamma, \Delta \vdash U \leq [Y/A]T\end{array}$$

and EQ-SUBSUMPTION.

The following three structural cases use straightforward equality reasoning. The final case, where T is an abstraction, cannot occur because T is of kind *Type*. □

B.9 Definition [c.f. Coquand and Paulin-Mohring (1989)]: When $pos(N)$,

$$\begin{array}{rcl} map_N & = & fun(X)\ fun(Y)\ fun(f{:}X{\rightarrow}Y)\ lift_S^{A\leftarrow f} \\ & : & All(X)\ All(Y)\ (X{\rightarrow}Y) \rightarrow (N(X){\rightarrow}N(Y)),\end{array}$$

where S is the β-normal form of $N(A)$.

B.10 Corollary: The definition of *map* satisfies the laws at the beginning of this section.

Proof: From B.7 and B.8, using EQ-SUBSUMPTION, S-CONV, and EQ-ABS to deal with the conversion to normal form in the definition of *map*. □

C Object Types

Having defined *pos* and *map*, we can further enrich $F^{\omega}_{\leq}$ with the high-level type constructor *Object* and the term constructors *object* and *GM*. Their associated typing, subtyping, and equational rules (taken from the text) are as follows:

$$\frac{\Gamma \vdash N \; : \; Type \rightarrow Type}{\Gamma \vdash Object(N) \; : \; Type} \tag{K-Obj}$$

$$\frac{\Gamma \vdash N' \leq N \qquad \Gamma \vdash N \; : \; Type \rightarrow Type}{\Gamma \vdash Object(N') \leq Object(N)} \tag{S-Obj}$$

$$\frac{\begin{array}{c}\Gamma \vdash N \; : \; Type \rightarrow Type \qquad pos(N) \\ \Gamma \vdash s \; : \; R \qquad \Gamma \vdash m \; : \; R \rightarrow N(R)\end{array}}{\Gamma \vdash object_N\,(R,\, s,\, m) \; : \; Object(N)} \tag{T-Obj-I}$$

$$\frac{\Gamma \vdash N \; : \; Type \rightarrow Type \qquad pos(N)}{\Gamma \vdash GM_N \; : \; All(N' \leq N) \;\; Object(N') \rightarrow N(Object(N'))} \tag{T-GM}$$

$$\frac{\begin{array}{c}\Gamma \vdash N' \leq N \qquad pos(N') \qquad pos(N) \\ \Gamma \vdash s \; : \; R \qquad \Gamma \vdash m \; : \; R \rightarrow N'(R)\end{array}}{\Gamma \vdash object_{N'}\,(R,\, s,\, m) = object_N\,(R,\, s,\, m) \; : \; Object(N)} \tag{Eq-Obj-Sub}$$

$$\frac{\Gamma \vdash N'' \leq N' \leq N \; : \; Type \rightarrow Type \qquad pos(N') \qquad pos(N)}{\Gamma \vdash GM_{N'}\,[N''] = GM_N\,[N''] \; : \; Object(N'') \rightarrow N(Object(N''))} \tag{Eq-GM-Sub}$$

$$\frac{\begin{array}{c}\Gamma \vdash N \; : \; Type \rightarrow Type \qquad pos(N) \\ \Gamma \vdash s \; : \; R \qquad \Gamma \vdash m \; : \; R \rightarrow N(R)\end{array}}{\begin{array}{l}\Gamma \vdash GM_N\,[N]\,(object_N\,(R,\, s,\, m)) \\ \quad = map_N\,[R]\,[Object(N)]\,(object_N\,(R,\, \text{—},\, m))\,(m\; s) \\ \quad : \;\; N(Object(N))\end{array}} \tag{Eq-Obj-Map}$$

C.1 Remark: Observe that by Eq-TBeta and Eq-Beta, we have

$$\begin{array}{l}\Gamma \vdash object_N\,(R,\, s,\, m) \\ \quad = (fun(R)\;\; fun(s{:}R)\;\; fun(m{:}R \rightarrow N(R))\;\; object_N\,(R,\, s,\, m))\,[R]\; s\; m \\ \quad : \;\; Object(N)\end{array}$$

From this equivalence and the equational rules in Section A.8 (Eq-TApp in particular), we may obtain a stronger version of Eq-Obj-Sub where R, s, and m are also permitted to vary.

D Existential Types

The sets of $F^{\omega}_{\leq}$ types and terms are extended with existentials as follows:

T	$::=$	$\ldots$	
	$\mid$	$Some(A{\leq}T_1)\,T_2$	existentially quantified type
e	$::=$	$\ldots$	
	$\mid$	$pack\; e\;\; as\; T_1\;\; hiding\; T_2$	packing
	$\mid$	$open\; e_1\;\; as\; [A,\, x]\;\; in\; e_2$	unpacking

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

$$\frac{\Gamma,\, A{\le}T_1 \vdash T_2 \;:\; \mathit{Type}}{\Gamma \vdash \mathit{Some}\,(A{\le}T_1)\,T_2 \;:\; \mathit{Type}} \tag{K-Some}$$

$$\frac{\begin{array}{c}\Gamma,\, A{\le}U \vdash S_2 \le T_2\\ \Gamma \vdash \mathit{Some}\,(A{\le}U)\,S_2 \;:\; \mathit{Type}\end{array}}{\Gamma \vdash \mathit{Some}\,(A{\le}U)\,S_2 \le \mathit{Some}\,(A{\le}U)\,T_2} \tag{S-Some}$$

$$\frac{\Gamma \vdash T \sim \mathit{Some}\,(A{\le}U_1)\,U_2 \qquad \Gamma \vdash S \le U_1 \qquad \Gamma \vdash e \;:\; [S/A]U_2}{\Gamma \vdash \mathit{pack}\ e\ \mathit{as}\ T\ \mathit{hiding}\ S \;:\; T} \tag{T-Some-I}$$

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

The equational theory is extended with the following rules (c.f. Martin-Löf's weak Σ-types (Smith *et al.*, 1990)):

$$\frac{\begin{array}{c}\Gamma \vdash T \sim \mathit{Some}\,(A{\le}U_1)\,U_2 \qquad \Gamma \vdash S \le U_1\\ \Gamma \vdash e = e' \;:\; [S/A]U_2\\ \Gamma,\, A{\le}U_1,\, x{:}U_2 \vdash b = b' \;:\; V\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{open}\ (\mathit{pack}\ e\ \mathit{as}\ T\ \mathit{hiding}\ S)\ \mathit{as}\ [A,\, x]\ \mathit{in}\ b\\ \quad = [e'/x][S/A]b'\\ \quad : V\end{array}} \tag{Eq-Some-Beta}$$

$$\frac{\begin{array}{c}\Gamma \vdash T' \sim \mathit{Some}\,(A{\le}U_1)\,U_2' \qquad \Gamma \vdash T \sim \mathit{Some}\,(A{\le}U_1)\,U_2\\ \Gamma \vdash T' \le T\\ \Gamma \vdash S', S \le U_1 \qquad \Gamma \vdash V \le [S'/A]U_2',\, [S/A]U_2'\\ \Gamma \vdash e' = e \;:\; V\end{array}}{\Gamma\vdash \mathit{pack}\ e'\ \mathit{as}\ T'\ \mathit{hiding}\ S' = \mathit{pack}\ e\ \mathit{as}\ T\ \mathit{hiding}\ S \;:\; T} \tag{Eq-Pack}$$

$$\frac{\begin{array}{c}\Gamma \vdash e = e' \;:\; \mathit{Some}\,(A{\le}T_1)\,T_2\\ \Gamma,\, A{\le}S_1,\, x{:}S_2 \vdash b = b' \;:\; T\end{array}}{\Gamma\vdash \mathit{open}\ e\ \mathit{as}\ [A,\, x]\ \mathit{in}\ b = \mathit{open}\ e'\ \mathit{as}\ [A,\, x]\ \mathit{in}\ b' \;:\; T} \tag{Eq-Open}$$

$$\frac{\begin{array}{c}\Gamma \vdash V \;:\; \mathit{Type}\\ \Gamma,\, y{:}\mathit{Some}\,(A{\le}S)\,T \vdash e,\, e' \;:\; V\\ \Gamma,\, A{\le}S,\, x{:}T \vdash [(\mathit{pack}\ x\ \mathit{as}\ \mathit{Some}\,(A{\le}S)\,T\ \mathit{hiding}\ S)/y](e = e') \;:\; V\end{array}}{\Gamma,\, y{:}\mathit{Some}\,(A{\le}S)\,T \vdash e = e' \;:\; V} \tag{Eq-Some-Ind}$$

D.1 Remark: All of the above rules except for the "induction principle" Eq-Some-Ind are valid under the usual encoding of existential types in terms of universal quantifiers:

$$\mathit{Some}\,(A{\le}S)\,T \quad = \quad \mathit{All}(B)\ \ (\mathit{All}(A{\le}S)\,T{\to}B) \to B.$$

Moreover, all the rules *including* Eq-Some-Ind are sound in the PER model, if we interpret the existential type as the sub-PER of the interpretation of the encoding restricted to those elements semantically equal to an element interpreting an expression of the form *pack....* More precisely, we define $e[\![Some(A{\leq}S)\,T]\!]_\eta e'$ iff $e[\![All(B)\ (All(A{\leq}S)\,T{\rightarrow}B) \rightarrow B]\!]_\eta e'$ and $e[\![All(B)\ (All(A{\leq}S)\,T{\rightarrow}B) \rightarrow B]\!]_\eta \lambda h.\, h\ x$, where $x : \mathrm{dom}([\![T]\!]_{\eta[A\leftarrow R]})$ for some PER $R \subseteq [\![S]\!]_\eta$. It also seems possible to model an existential by the transitive closure of the union of all instances of its body. It is a matter for future research to inquire whether the equational theory of existential types can be axiomatized using only unconditional equations. Interestingly, Lemma D.5 can also be proven for the universal encoding of existentials.

D.2 Definition: The definitions of the predicates pos_A and neg_A (Definition B.2) are extended as follows:

$$\begin{array}{lcl} pos_A(Some(B{\leq}T_1)\,T_2) & = & A \text{ not free in } T_1 \text{ and } pos_A(T_2) \\ neg_A(Some(B{\leq}T_1)\,T_2) & = & A \text{ not free in } T_1 \text{ and } neg_A(T_2) \end{array}$$

D.3 Fact: These definitions validate the rule S-Pos-Mono for $F^\omega_\leq$ extended with existentials.

Proof: Extend the proof of B.4 with an analogous case for existentials, using S-Some. □

D.4 Definition: The definitions of the lifting and lowering of a function f through a type T (B.6) are extended as follows:

$$\begin{array}{lcl} lift_{Some(B\leq T_1)T_2} & = & fun(g : (Some(B{\leq}T_1)\ [X/A]T_2)) \\ & & \quad open\ g\ \ as\ [B, x]\ in \\ & & \qquad pack\ (lift_{T_2}\ x) \\ & & \qquad as\ Some(B{\leq}T_1)\ \ [Y/A]T_2 \\ & & \qquad hiding\ B \\ & & \text{when } A \text{ is not free in } T_1 \\ lower_{Some(B\leq T_1)T_2} & = & fun(g : (Some(B{\leq}T_1)\ [Y/A]T_2)) \\ & & \quad open\ g\ \ as\ [B, x]\ in \\ & & \qquad pack\ (lower_{T_2}\ x) \\ & & \qquad as\ Some(B{\leq}T_1)\ \ [X/A]T_2 \\ & & \qquad hiding\ B \\ & & \text{when } A \text{ is not free in } T_1 \end{array}$$

D.5 Lemma: The rules listed at the beginning of Section B are derivable in the extended calculus.

Proof: By extension of the previous inductive arguments, leading up to Corollary B.10. For Eq-Map-Id and Eq-Map-Trans, we use Eq-Some-Ind to "replace" the variable g in the definition of *lift* and *lower* by an instance of *pack*. The result then follows by straightforward equality reasoning using Eq-Some-Beta. For Eq-Map-Sub, we use Eq-Pack and Eq-Open. □

E Recursive Types

The following extension of the basic $F^\omega_\leq$ calculus with recursive types is somewhat tentative. The rules are suggested by existing treatments of recursive types in lower-order calculi (Amadio & Cardelli, 1993), but a full study of recursive types in this setting falls outside the scope of the present article.

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 formation rule

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

and two subtyping rules — one for "unfolding" a recursive type and one for (finitely) comparing two recursive types (c.f. (Amadio & Cardelli, 1993)):

$$\frac{\Gamma \vdash \mu(A{:}K)T \;:\; K}{\Gamma \vdash \mu(A{:}K)T \sim [(\mu(A{:}K)T)/A]T} \tag{S-Fold}$$

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

E.1 Remark: Note that we have the derived rule

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

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 the following extension of the definitions of *pos* and *neg*.

E.2 Definition: The definitions of the predicates pos_A and neg_A (B.2) are extended as follows:

$$\begin{array}{lcl} \mathit{pos}_A(\mu(B)T_2) & = & \mathit{pos}_A(T_2) \text{ and } \mathit{pos}_B(T_2) \\ \mathit{neg}_A(\mu(B)T_2) & = & \mathit{neg}_A(T_2) \text{ and } \mathit{pos}_B(T_2) \end{array}$$

The definition of the *pos* predicate still makes sense, since, for purposes of type normalization, the μ operator can simply be treated as a constant.

E.3 Remark: Note that a type variable A appears positively in a type $T = \mu(B)F$ only if *both* A and B appear only positively in F.

E.4 Definition: We introduce a value-level fixed-point combinator *fix* with the typing rule

$$\mathit{fix} \;:\; \mathit{All}(A{\leq}\mathit{Top}(\mathit{Type}))\;\; (A{\rightarrow}A) \rightarrow A \tag{T-Fix}$$

and (in addition to the equations implied by its typing), the equational rules

$$\frac{\Gamma \vdash f \;:\; A{\rightarrow}A}{\Gamma \vdash \mathit{fix}\;[A]\;f = f\;(\mathit{fix}\;[A]\;f) \;:\; A} \tag{Eq-Fix}$$

$$\frac{\begin{array}{c}\Gamma \vdash A' \leq A \\ \Gamma \vdash f' \;:\; A'{\rightarrow}A' \qquad \Gamma \vdash f \;:\; A{\rightarrow}A \\ \Gamma \vdash f = f' \;:\; A'{\rightarrow}A\end{array}}{\Gamma \vdash \mathit{fix}\;[A]\;f = \mathit{fix}\;[A']\;f' \;:\; A} \tag{Eq-Fix-Sub}$$

which respectively characterize *fix* as a fixed point and describe its behavior with respect to subtyping.

E.5 Remark: These rules can be interpreted in the "cuper model" of Amadio and Cardelli (1993), realizing *fix* by $\lambda f.\ \bigsqcup_{k\,:\,\omega} f^k \bot$. For EQ-FIX-SUB, we use the observation that $f = f' \ :\ A' \rightarrow A$ implies $f^k = f'^k \ :\ A' \rightarrow A$ (by nontrivial equality reasoning), and then use continuity.

Amadio and Cardelli actually exhibit a typed version of the usual Y combinator satisfying T-FIX and EQ-FIX. Interestingly, EQ-FIX-SUB does not seem to be provable for this term. Our *fix* therefore constitutes a proper extension of their calculus.

E.6 Definition: The definitions of the lifting and lowering of a function f through a type T (B.6) are extended as follows. To reduce clutter, we adopt the abbreviation $S[U,V] = [V/B][U/A]S$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{lift}_{\mu(B)S} =} \\
\multicolumn{2}{l}{\quad \mathit{fix}\ [\mu(B)S[X,B] \rightarrow \mu(B)S[Y,B]]} \\
\multicolumn{2}{l}{\qquad \mathit{fun}\,(g : \mu(B)\ S[X,B] \rightarrow \mu(B)S[Y,B])} \\
\qquad\quad (\mathit{unfold} : & \mu(B)S[X,B] \rightarrow S[X,\mu(B)S[X,B]]; \\
\qquad\quad (\mathit{lift}^{A \leftarrow f}_{S[A,\mu(B)S[X,B]]} : & S[X,\mu(B)S[X,B]] \rightarrow S[Y,\mu(B)S[X,B]]); \\
\qquad\quad (\mathit{lift}^{B \leftarrow g}_{S[Y,B]} : & S[Y,\mu(B)S[X,B]] \rightarrow S[Y,\mu(B)S[Y,B]]); \\
\qquad\quad (\mathit{fold} : & S[Y,\mu(B)S[Y,B]] \rightarrow \mu(B)S[Y,B]) \\
\multicolumn{2}{l}{\quad \text{when } \mathit{pos}_A(S) \text{ and } \mathit{pos}_B(S)}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{lower}_{\mu(B)S} =} \\
\multicolumn{2}{l}{\quad \mathit{fix}\ [\mu(B)S[Y,B] \rightarrow \mu(B)S[X,B]]} \\
\multicolumn{2}{l}{\qquad \mathit{fun}\,(g : \mu(B)\ S[Y,B] \rightarrow \mu(B)S[X,B])} \\
\qquad\quad (\mathit{unfold} : & \mu(B)S[Y,B] \rightarrow S[Y,\mu(B)S[Y,B]]; \\
\qquad\quad \mathit{lower}^{A \leftarrow f}_{S[A,\mu(B)S[Y,B]]} : & S[Y,\mu(B)S[Y,B]] \rightarrow S[X,\mu(B)S[Y,B]]); \\
\qquad\quad (\mathit{lift}^{B \leftarrow g}_{S[X,B]} : & S[X,\mu(B)S[Y,B]] \rightarrow S[X,\mu(B)S[X,B]]); \\
\qquad\quad (\mathit{fold} : & S[X,\mu(B)S[X,B]] \rightarrow \mu(B)S[X,B]) \\
\multicolumn{2}{l}{\quad \text{when } \mathit{neg}_A(S) \text{ and } \mathit{pos}_B(S)}
\end{array}
$$

(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 would be inferred by a typechecker.)

E.7 Remark: In order to show that the extended definition of *map* still satisfies the laws in Appendix B, we need 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), the laws seem to hold 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).

References

Abadi, M. 1992 (Feb.). *Doing without F-bounded quantification.* Message to `Types` electronic mail list.

Abadi, M. 1994. Baby Modula-3 and a Theory of Objects. *Journal of Functional Programming*, **4**(2). An earlier version appeared as DEC Systems Research Center Research Report 95, (February, 1993).

Abadi, M., & Cardelli, L. 1994b. A Theory of Primitive Objects: Second-order Systems. *In: European Symposium on Programming (ESOP), Edinburgh, Scotland.*

Abadi, M., & Cardelli, L. 1994a. A Theory of Primitive Objects: Untyped and First-order Systems. *In: Theoretical Aspects of Computer Software (TACS), Sendai, Japan.*

Amadio, R. M., & Cardelli, L. 1993. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems*, **15**(4), 575–631. A preliminary version appeared in POPL '91 (pp. 104–118), and as DEC Systems Research Center Research Report number 62, August 1990.

Barendregt, H. 1992. Lambda Calculi with Types. *In:* Abramsky, G., & Maibaum (eds), *Handbook of Logic in Computer Science*, vol. II. Oxford University Press.

Barr, M., & Wells, C. 1990. *Category Theory for Computing Science.* Prentice Hall.

Bruce, K., & Mitchell, J. 1992 (Jan.). PER models of subtyping, recursive types and higher-order polymorphism. *In: Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages.*

Bruce, K. B. 1994. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, **4**(2). A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

Canning, P., Cook, W., Hill, W., Olthoff, W., & Mitchell, J. 1989 (Sept.). F-Bounded Quantification for Object-Oriented Programming. *Pages 273–280 of: Fourth International Conference on Functional Programming Languages and Computer Architecture.*

Cardelli, L. 1984. A semantics of multiple inheritance. *Pages 51–67 of:* Kahn, G., MacQueen, D., & Plotkin, G. (eds), *Semantics of Data Types.* Lecture Notes in Computer Science, vol. 173. Springer-Verlag. Full version in *Information and Computation* 76:138–164, 1988.

Cardelli, L. 1988 (Jan.). Structural Subtyping and the Notion of Power Type. *Pages 70–79 of: Proceedings of the 15th ACM Symposium on Principles of Programming Languages.*

Cardelli, L. 1990 (Oct.). *Notes about* $F^{\omega}_{<:}$. Unpublished manuscript.

Cardelli, L. 1992 (Jan.). *Extensible Records in a Pure Calculus of Subtyping.* Research report 81. DEC Systems Research Center. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Cardelli, L., & Longo, G. 1991. A semantic basis for Quest. *Journal of Functional Programming*, **1**(4), 417–458. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

Cardelli, L., & Wegner, P. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, **17**(4).

Cardelli, L., Martini, S., Mitchell, J. C., & Scedrov, A. 1994. An Extension of System F with Subtyping. *Information and Computation*, **109**(1–2), 4–56. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

Castagna, G., Ghelli, G., & Longo, G. 1994. A calculus for overloaded functions with subtyping. *Information and Computation.* To appear; a preliminary version appeared in LISP and Functional Programming, July 1992 (pp. 182–192), and as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

Compagnoni, A. B. 1994 (Jan.). *Subtyping in $F^{\omega}_{\wedge}$ is decidable.* Tech. rept. ECS-LFCS-94-281. LFCS, University of Edinburgh. To appear in the proceedings of Computer Science Logic, September 1994, under the title "Decidability of Higher-Order Subtyping with Intersection Types".

Compagnoni, A. B., & Pierce, B. C. 1993 (Aug.). *Multiple Inheritance via Intersection Types.* Tech. rept. ECS-LFCS-93-275. LFCS, University of Edinburgh. Also available as Catholic University Nijmegen computer science technical report 93-18.

Cook, W. 1991. Object-oriented programming versus abstract data types. *Pages 151–178 of:* de Bakker, J. W., *et al.*(eds), *Foundations of Object-Oriented Languages.* Lecture Notes in Computer Science, vol. 489. Springer-Verlag.

Cook, W. 1989. *A Denotational Semantics of Inheritance.* Ph.D. thesis, Brown University.

Cook, W. R., Hill, W. L., & Canning, P. S. 1990 (Jan.). Inheritance is not Subtyping. *Pages 125–135 of: Seventeenth Annual ACM Symposium on Principles of Programming Languages.* Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Coquand, T., & Paulin-Mohring, C. 1989. Inductively defined types. *In: LNCS 389.* Springer-Verlag.

Danforth, S., & Tomlinson, C. 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, **20**(1), 29–72.

Fisher, K., & Mitchell, J. 1994. Notes on Typed Object-Oriented Programming. *Pages 844–885 of: Proceedings of Theoretical Aspects of Computer Software, Sendai, Japan.* Springer-Verlag. LNCS 789.

Girard, J.-Y. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. thesis, Université Paris VII.

Goldberg, A., & Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley.

Hofmann, M., & Pierce, B. 1992. *An Abstract View of Objects and Subtyping (Preliminary Report).* Technical Report ECS-LFCS-92-226. University of Edinburgh, LFCS.

Hofmann, M., & Pierce, B. 1994 (Sept.). *Positive Subtyping.* Tech. rept. ECS-LFCS-94-303. LFCS, University of Edinburgh.

Kamin, S. N., & Reddy, U. S. 1994. Two Semantic Models of Object-Oriented Languages. *Pages 464–495 of:* Gunter, C. A., & Mitchell, J. C. (eds), *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press.

Kock, A. 1970. Strong Functors and Monoidal Monads. *Various Publications Series 11, Aarhus Universitet.*

Läufer, K., & Odersky, M. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Transactions on Programming Languages and Systems (TOPLAS).* To appear; an earlier version appeared in the Proceedings of the ACM SIGPLAN Workshop on ML and its Applications, 1992, under the title "An Extension of ML with First-Class Abstract Types".

Mitchell, J., & Plotkin, G. 1988. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, **10**(3).

Mitchell, J. C. 1990a (Jan.). Toward a Typed Foundation for Method Specialization and Inheritance. *Pages 109–124 of: Proceedings of the 17th ACM Symposium on Principles of Programming Languages.* Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Mitchell, J. C. 1990b. A Type-Inference Approach to Reduction Properties and Semantics of Polymorphic Expressions. *Pages 195–212 of:* Huet, G. (ed), *Logical Foundations of Functional Programming.* University of Texas at Austin Year of Programming Series. Addison-Wesley.

Moggi, E. 1989. Computational lambda-calculus and monads. *Pages 14–23 of: Fourth Annual Symposium on Logic in Computer Science (Asilomar, CA).* IEEE Computer Society Press.

Pierce, B., Dietzen, S., & Michaylov, S. 1989 (Mar.). *Programming in Higher-order Typed Lambda-Calculi.* Technical Report CMU-CS-89-111. Carnegie Mellon University.

Pierce, B. C., & Turner, D. N. 1993 (Apr.). *Statically Typed Friendly Functions via Partially Abstract Types.* Technical Report ECS-LFCS-93-256. University of Edinburgh, LFCS. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

Pierce, B. C., & Turner, D. N. 1994. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming,* **4**(2), 207–247. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

Reichel, H. 1995. An Approach to Object Semantics based on Terminal Co-algebras. *Mathematical Structures in Computer Science.* To appear.

Reynolds, J. 1974. Towards a Theory of Type Structure. *Pages 408–425 of: Proc. Colloque sur la Programmation.* New York: Springer-Verlag LNCS 19.

Reynolds, J. C. 1978. User Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. *Pages 309–317 of:* Gries, D. (ed), *Programming Methodology, A Collection of Articles by IFIP WG2.3.* New York: Springer-Verlag. 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. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Smith, J., Nordström, B., & Petersson, K. 1990. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press.

Steffen, M., & Pierce, B. 1994 (June). Higher-Order Subtyping. *In: IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET).* An earlier version appeared as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

Stroustrup, B. 1986. *The C++ Programming Language.* Reading, Mass: Addison-Wesley.

Wand, M. 1987 (June). Complete type inference for simple objects. *In: Proceedings of the IEEE Symposium on Logic in Computer Science.*

Wraith, G. C. 1989. A note on categorical datatypes. Lecture Notes in Computer Science, no. 389. Springer-Verlag.