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

Simple Type-Theoretic Foundations For Object-Oriented Programming

Benjamin C. Pierce David N. Turner

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

Abstract

We develop a formal, type-theoretic account of the basic mechanisms of object-oriented programming: encapsulation, message passing, subtyping, and inheritance. By modeling object encapsulation in terms of existential types instead of the recursive records used in other recent studies, we obtain a substantial simplification both in the model of objects and in the underlying typed λ-calculus.

1 Introduction

Static type systems for object-oriented programming languages have progressed significantly in the past decade. The line of research begun by Cardelli (1988a), Cook (1989), Reddy (1988), and Kamin (1988) and further developed by Cardelli (Cardelli & Wegner, 1985; Cardelli & Mitchell, 1991; Cardelli, 1992a; Cardelli, 1990), Mitchell (1990; 1989; 1991), Bruce (1992; 1991; 1990), Wand (1987; 1988; 1989), and many others (Canning *et al.*, 1989; Cook *et al.*, 1990; Castagna *et al.*, 1992; Castagna, 1992; Ghelli, 1991; Graver & Johnson, 1990) has culminated in type-theoretic accounts (Bruce, 1992; Cardelli, 1992a; Mitchell *et al.*, 1993) of many of the features of languages like Smalltalk (Goldberg & Robson, 1983). Our goal is to reformulate the essential ideas of these accounts using a simpler type theory.

The key step in our approach is an alternative treatment of encapsulation. Reynolds (1978) identified two complementary approaches to encapsulation: *procedural abstraction*, which relies on hiding state in private variables common to a collection of procedures, and *type abstraction*, which reveals the existence of state externally but prevents illegal access to it by hiding its type. Previous accounts of object-oriented programming have chosen procedural abstraction, encoding objects as elements of recursive record types. For example, the type of movable, one-dimensional point objects is usually encoded as:

$$Point = Rec(P)\ \{\!|getX : Int,\ setX : Int{\to}P|\!\}$$

We choose type abstraction instead, following a close analogy with Mitchell and Plotkin's treatment of conventional abstract types as existential types (1988):

$$Point = \exists(Rep)\ \{\!|state : Rep,\ methods : \{\!|getX : Rep{\to}Int,\ setX : Rep{\to}Int{\to}Rep|\!\}\,|\!\}$$

Both the state of an object (the component *state*) and the methods operating on it are visible in this encoding, with the existential type protecting the state from external access.

The principal benefit of this change is a simplification in the underlying type theory: it allows us to give a complete model of encapsulation, message passing, subtyping, and inheritance (including the special names `self` and `super`) using neither recursive types — which until now have generally been regarded as essential (Bruce, 1992; Mitchell, 1990) — nor the sophisticated record extension operations that appear in some accounts. Moreover, the naïveté of our basic encoding yields a clear separation between the simple aspects of the object model, encapsulation and subtyping, and other, more complex but less essential features such as inheritance. By regarding inheritance as "just a matter of programming" in terms of the constructions provided by the basic object model, we can develop and compare several alternative implementations, including an implementation of polymorphic classes, a useful extension that has not yet been attempted in other models. The expressive power of our model of objects is comparable to previous models; indeed, it appears that the only significant difference is that our model requires a different treatment of binary methods (c.f. Section 10).

Section 2 develops our encoding of objects in more detail. In Section 3 we introduce subtyping. Proper treatment of the interaction between encapsulation and subtyping requires a mechanism for expressing refined subtyping constraints, which we obtain by extending subtyping to type operators. Section 4 sketches a simple high-level syntax for object type declarations with a uniform translation into the basic calculus. Section 5 introduces the basic concepts of inheritance, using a simple high-level syntax for class definitions. In Sections 6 and 7 we show in detail how two different forms of inheritance can be implemented within the framework of our basic object model. Section 8 develops a longer example, a functional variant of two of the Smalltalk collection classes, illustrating a more interesting use of inheritance; this example is generalized to polymorphic collections in Section 9 with yet another implementation of inheritance. In Section 10 we compare our work to other proposed static type systems for object-oriented programming and survey some extensions of our basic model.

Our model of objects is given in $F^{\omega}_{\leq}$, a higher-order explicitly-typed λ-calculus with subtyping. A short introduction to $F^{\omega}_{\leq}$ is given in Appendix A. Appendix B summarizes the syntax and typing rules for easy reference. The examples in the paper were typeset using a prototype compiler for $F^{\omega}_{\leq}$ that typechecks and evaluates declarations preceded by the symbol `#`. Declarations may be split across a number of lines, and are terminated using a semicolon. A `%` symbol indicates that the rest of the line is a comment and will be ignored by the compiler.

Basic familiarity with polymorphic type systems, subtypes, existential types, and conventional object-oriented languages will be helpful for understanding this paper; background reading on these topics can be found in (Mitchell & Plotkin, 1988; Cardelli & Wegner, 1985; Cardelli, 1988a; Budd, 1991; Goldberg & Robson, 1983; Reynolds, 1985).

2 Objects

As in most type-theoretic accounts of object-oriented programming, we restrict our attention to purely functional objects, in which methods must return a new copy of the internal state instead of updating it in place. (Questions of typing are not affected by this simplification; the model can straightforwardly be extended to include imperative-style object-oriented programming (Bruce & van Gent, 1993), modulo one technical proviso; c.f. Section 10.) The state of an object is represented by a single value. For example, the state of a one-dimensional point object with x-coordinate `5` is the one-field record

```
# {x=5};
<val> : {|x: Int|}
```

A method is a function that implements some transformation on the state. For example, a `bump` method for point objects might return a state whose `x`-coordinate has been increased by one:

```
# bump = fun(state:{|x: Int|}) {x = plus 1 state.x};
bump = <val> : {|x:Int|} -> {|x:Int|}
```

A `setX` method takes an extra parameter, which becomes the `x`-coordinate of the new state:

```
# setX = fun(state:{|x: Int|}) fun(newX: Int) {x = newX};
setX = <val> : {|x:Int|} -> Int -> {|x:Int|}
```

Instead of returning the new state for an object, a method may extract some other information. For example, the `getX` method returns the current `x`-coordinate:

```
# getX = fun(state:{|x: Int|}) state.x;
getX = <val> : {|x:Int|} -> Int
```

Since the internal state of a Smalltalk-style object is accessible only to its methods, the object's interface to the outside world can be expressed by replacing the type of the state by an abstract token `Rep` in the types of its methods:

```
bump: Rep->Rep
setX: Rep->Int->Rep
getX: Rep->Int
```

Formally, this replacement is accomplished by regarding the type of the methods as a *function* from the representation type to the type of a record of functions:

```
# PointM = Fun(Rep) {|
#   bump: Rep->Rep,
#   setX: Rep->Int->Rep,
#   getX: Rep->Int
# |};
PointM : *->*
```

This function can be applied to any particular representation of points, such as the record type `{|x:Int|}`, to yield the types of the methods of objects based on that representation.

An object satisfying the above specification consists of a record of methods of type `PointM Rep` for some concrete state type `Rep`, paired with a "current state" of type `Rep` and surrounded by an abstraction barrier that protects the current state from access except through the methods. This encapsulation is directly expressed by an existential type:

```
# Point = Some(Rep) {| state: Rep, methods: PointM Rep |};
Point : *
```

Abstracting `PointM` from `Point` yields a higher-order type operator that, given an interface specification, forms the type of objects satisfying it:

```
# Object = Fun(M:*->*) Some(Rep) {| state: Rep, methods: M Rep |};
Object : (*->*)->*
```

Here `*` is the kind of well-formed types such as `Int` and `Int->Int` and `*->*` is the kind of functions from types to types, such as `Fun(X:*)X->X`. Since `Object` takes an argument of kind `*->*` and returns a type, its own kind is `(*->*)->*`. The type of point objects can now be expressed more concisely by applying the `Object` constructor to the specification `PointM`:

```
# Point = Object PointM;
Point : *
```

At this stage, the separation of `Point` into a specification of its methods and an operator capturing the common structure of all object types is just a matter of notational convenience. This separation will be crucial, however, for handling the interaction between encapsulation and subtyping.

New point objects are created using the existential introduction form `<R,r>:T`, which packages the witness type `R` and the body `r` into an element of the existential type `T`. For example, a point object with representation type `{|x: Int|}`, internal state `{x=5}`, and method implementations as above can be created as follows:

```
# p1 = < {|x: Int|},
#        {state = {x=5},
#         methods = {bump = fun(s:{|x:Int|}) {x=plus 1 s.x},
#                    setX = fun(s:{|x:Int|}) fun(i: Int) {x=i},
#                    getX = fun(s:{|x:Int|}) s.x}}
#      >: Point;
p1 = <val> : Point
```

Note that, unlike some object-oriented languages, the elements of an object type here may have different internal representations and different implementations of their methods. For example, a point with representation type `{|x:Int,other:Int|}` might be implemented as follows:

```
# p2 = < {|x: Int, other: Int|},
#        {state = {x=5,other=999},
#         methods = {bump = fun(s:{|x:Int,other:Int|})
#                             {x=plus 1 s.x,other=s.other},
#                    setX = fun(s:{|x:Int,other:Int|}) fun(i:Int)
#                             {x=i,other=s.other},
#                    getX = fun(s:{|x:Int,other:Int|})
#                             s.x}}
#      >: Point;
p2 = <val> : Point
```

The internal `bump` method of an arbitrary point `p` can be invoked using the following message-sending function:

```
# Point'bump =
#  fun(p: Point)
#    open p as <Rep,r> in
#      < Rep, {state = r.methods.bump r.state, methods = r.methods} >: Point
#    end;
Point'bump = <val> : Point -> Point
```

First we open `p`, binding the variable `Rep` to its representation type and the record containing its state and methods to the variable `r`. The typechecking rule for `open` ensures that the representation type can only be used abstractly in the body of the `open`: the only functions applicable to the state component `r.state` are those in `r.methods`. We apply the `bump` function from `r.methods` to `r.state`, producing a new value of type `Rep`, which is used to create a new point object (having the same methods and hidden representation type as the original). The name `Point'bump` introduces a convention that we will follow throughout: `T'm` names the function that sends the message `m` to objects of type `T = Object TM`.

The `Point'setX` function is implemented in almost exactly the same way; the only difference is that we need to add an extra parameter `i`, the new `x`-coordinate for the point, and then apply the function `setX` from `r.methods` to both `r.state` and `i`:

```
# Point'setX =
#   fun(p: Point)
#     open p as <Rep,r> in
#       fun(i: Int)
#         < Rep, {state = r.methods.setX r.state i, methods = r.methods}
#         >: Point
#     end;
Point'setX = <val> : Point -> Int -> Point
```

Since the `getX` method does not return a new state, we don't need to create a new point object; we just apply the `getX` function from `r.methods` to the state to yield an integer, which we then return:

```
# Point'getX =
#   fun(p: Point)
#     open p as <Rep,r> in
#       r.methods.getX r.state
#     end;
Point'getX = <val> : Point -> Int
```

Now we can write programs that send the messages `setX`, `getX`, and `bump` to point objects:

```
# Point'getX (Point'bump (Point'setX p1 3));
4 : Int
```

The conventional use of existential types in modeling abstract types differs from their use here in modeling objects, in that a "package" (an instance of an existential type) implementing an abstract type is normally opened just once, as soon as it is created, whereas a package representing an object is kept closed until the last possible moment, when its internal state and methods must be combined in the body of a message-sending function.

In the recursive-records encoding of objects, sending a message to an object is usually modeled by extracting a method from the record and applying it to any additional arguments. There, the responsibility for performing any unpacking and repacking of internal state is placed on the receiver of a message; we give the responsibility to the sender.

3 Subtyping

Most object-oriented languages organize the specifications of objects into a subtype hierarchy, capturing the intuition that some objects may provide more services than others. For example, consider a refined form of point objects that carry color as well as position information:

```
# CPointM = Fun(Rep) {| getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep,
#                       getC: Rep->Color, setC: Rep->Color->Rep |};
CPointM : *->*
# CPoint = Object CPointM;
CPoint : *
```

We would expect that an element of `CPoint` can safely be used in any context where a `Point` is expected. This intuition is formalized by extending our λ-calculus with a subtype relation, writing $\Gamma \vdash S \leq T$ to mean that S is a subtype of T under assumptions Γ. The subtype relation we use here is a straightforward higher-order extension (due to Cardelli (1990) and Mitchell (1990)) of the familiar calculus of second-order bounded quantification, $F_{\leq}$ (Cardelli & Wegner, 1985; Cardelli *et al.*, 1991; Curien & Ghelli, 1992).

Using the subtyping rules in Appendix B, it is easy to check that `CPoint` is indeed a subtype of `Point`, as follows. The definitions of `CPoint` and `Point` are equivalent (by β-conversion) to:

```
# CPoint = Some(Rep) {|
#   state: Rep,
#   methods: {| getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep,
#               getC: Rep->Color, setC: Rep->Color->Rep |}
# |};
CPoint : *
```

```
# Point  = Some(Rep) {|
#   state: Rep,
#   methods: {| getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep |}
# |};
Point : *
```

By the rules for existential and record subtyping (S-SOME and S-RECORD), `CPoint` $\leq$ `Point` if

```
{|getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep,
  getC: Rep->Color, setC: Rep->Color->Rep|}
```

$\leq$

```
{|getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep |},
```

which holds (by S-RECORD again) since the former has more fields and the types of the common fields agree.

The encoding of message passing in Section 2 must be generalized to interact properly with subtyping. For example, the typing of `Point'bump` already allows it to be applied to an element of `CPoint` (using rules T-SUBSUMPTION and T-ARROW-E), but the result of the application is an element of `Point`, rather than an element of `CPoint`. (We want the result to be of type `CPoint` since `Point'bump` is supposed to be a functional analog of the operation of sending the `bump` method to an object.) It would be unfortunate if sending such a message caused the sender to lose information about the type of the receiver.

Cardelli and Wegner (1985) proposed that the proper type for a function such as `Point'bump` should be:

```
Point'bump : All(P<Point) P->P
```

That is, given any subtype of the type of point objects, `Point'bump` should map elements of this type into results of the same type. This is an intuitively reasonable typing; unfortunately, it is not possible to generalize our implementation of `Point'bump` so that it possesses this typing.

(It is instructive to check that the "obvious" generalization

```
wrong'bump =
  fun(P<Point)
    fun(p: P)
      open p as <Rep,r> in
        < Rep, {state = r.methods.bump r.state, methods = r.methods} >: P
      end;
```

is not well typed. The typing rule for the `open` expression requires that the type of the expression being opened have the form `Some(R)T`, for some `T`; but here the type declared for `p` is `P`, which has the form of a variable rather than an existential quantifier. In order to use the `open` rule, we must apply the rule of subsumption to `p`, promoting its type to be `Point` (this is legal, since `P < Point`). But now, when we apply the internal `bump` function to the state and combine the result with the old record of methods, we are only justified in claiming to have built the internal state of a new `Point`, not the internal state of an element of `P`, and so the existential introduction `<Rep,...>:P` is ill typed. In effect, this is the same problem that we had before introducing the bounded quantifier: to use an element of `P` as a `Point`, the rule of subsumption must be applied, leading to an irrevocable loss of information.)

Indeed, a simple semantic argument (c.f. (Robinson & Tennent, 1988)) shows that, in some models (for example, those based on partial equivalence relations (Bruce & Longo, 1990)), the only inhabitants of the type `All(P<Point) P->P` are identity functions. To see this, imagine a subtype `P` of `Point` containing just one element. If we instantiate `Point'bump` with `P` and then pass the one element of `P` as its argument, the result can clearly only be the same point. That is, on this one-element type `P`, `Point'bump` behaves like an identity function. But since we are working in a λ-calculus whose notion of polymorphism is *parametric* (Reynolds, 1983), the behavior of polymorphic functions cannot depend on their type arguments. The fact that `Point'bump` is an identity function at one type implies that it is an identity function at all types.

To obtain a sound typing for `Point'bump`, we need to consider more carefully what we want the typing to express. It is not the case that we need to be able to apply `Point'bump` to elements of arbitrary subtypes of `Point`; it suffices that we be able to apply it to elements of arbitrary *object types* whose *interfaces* are more refined than the interface of point objects:

```
Point'bump : All(M<PointM) (Object M) -> (Object M)
```

Informally, it should be clear what is meant by the quantification `All(M<PointM)`. But we need to define what it means formally, in terms of the subtype relation, for one type operator to be a subtype of another. In fact, there are several reasonable alternatives; for present purposes it suffices to consider the simplest possible one: subtyping on types is simply extended *pointwise* to operators. An operator `M:*->*` is a subtype of `N:*->*` if, whenever `M` and `N` are instantiated with the same type `T`, the results stand in the subtype relation: `M T` $\leq$ `N T`. In particular, we say that `Fun(A:K)S` is a subtype of `Fun(A:K)T` if `S`$\leq$`T`. Hence, `CPointM` $\leq$ `PointM`, since we have already checked that

```
{|getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep,
  getC: Rep->Color, setC: Rep->Color->Rep|}
```

$\leq$

```
{|getX: Rep->Int, setX: Rep->Int->Rep, bump: Rep->Rep |}
```

for every possible instantiation of the type variable `Rep`.

Our earlier implementation can easily be generalized so that it possesses the required type:

```
# Point'bump =
#   fun(M<PointM)
#     fun(p: Object M)
#       open p as <Rep,r> in
#         < Rep, {state = r.methods.bump r.state,
#                 methods = r.methods}
#         >: Object M
```

```
#        end;
Point'bump = <val> : All(M<PointM) (Object M) -> (Object M)
```

Since CPointM $\leq$ PointM, we can apply this version of Point'bump to the type operator CPointM yielding a message-sending function which maps an element of CPoint to a CPoint, as desired:

```
# Point'bump CPointM;
<val> : (Object CPointM) -> (Object CPointM)
```

The polymorphic message sending functions for points can now be applied to colored points without losing type information; if cp is a colored point, we can write:

```
# Point'getX CPointM (Point'bump CPointM (Point'setX CPointM cp 3));
4 : Int
```

Intuitively, our use of operator subtyping permits the typing of message-sending operations to distinguish the updateable portions of a data structure, in which subtyping must not be allowed, from the portions that are never changed by the message-sending functions, where subtyping is permissible, even in the presence of type constructors such as existential quantifiers that introduce type variable bindings. Operator subtyping appears in many type theoretic accounts of object-oriented programming. Mitchell (1990) uses it explicitly, while Cook, Hill, and Canning (1990) and Bruce (1992; 1993) rely on the closely related formalism of F-bounded quantification to achieve a similar effect. Mitchell, Honsell, and Fisher's more recent type system for delegation-based inheritance (1993) uses a similar mechanism to ensure the soundness of their object extension operator.

4 High-level Syntax for Objects

We have presented our encodings in "bare" $F^{\omega}_{\leq}$ in order to be very precise about the type-theoretic treatment of the basic mechanisms of object-oriented programming. Of course, the constructions we have given are too verbose to be of direct use in practice. To alleviate this problem, our implementation of $F^{\omega}_{\leq}$ provides a concise high-level syntax for declaring object types and their associated message-sending functions.

Our high-level syntax for object types relies on the observation that message-sending functions like Point'setX can be generated uniformly from the types of the methods: given the ObjectType declaration below, we automatically generate the types PointM and Point and the implementations of Point'getX, Point'setX and Point'bump.

```
# Point =
#   ObjectType(Rep) with
#     getX: Int,
#     setX: Int->Rep,
#     bump: Rep
#   end;
PointM = Fun(Rep) {|getX: Rep->Int,  setX: Rep->Int->Rep,  bump: Rep->Rep|}
Point = Object PointM
```

```
Point'getX : All(M<PointM) (Object M) -> Int
Point'setX : All(M<PointM) (Object M) -> Int -> (Object M)
Point'bump : All(M<PointM) (Object M) -> (Object M)
```

Intuitively, the generation of these functions is quite straightforward: the compiler needs to find all the occurrences of the representation type `Rep` in the result types of the methods and insert the necessary re-packaging code to build new objects instead of returning a bare instance of `Rep`. For example, the implementation of `Point'bump` generated by the compiler is identical to that show in section 3. A theoretical justification of this compilation procedure is developed in detail in (Hofmann & Pierce, 1994).

The following declaration of `CPoint` generates functions for sending the `setX`, `getX`, `bump`, `setC` and `getC` messages. Here, the functions `CPoint'setX`, `CPoint'getX` and `CPoint'bump` are actually redundant, since the corresponding `Point` message-sending functions have essentially the same typing. In general, however, `CPoint`'s version of a given method may have a more specific type than `Point`'s. For example, the colored point `getX` method could have type `Pos` (where `Pos`, the type of positive numbers, is a subtype of `Int`). In this case, `CPoint'getX` would have type `All(M<CPointM) (Object M) -> Pos` and having both the `Point` and the `CPoint` message-sending functions would be useful.

```
# CPoint =
#   ObjectType(Rep) with
#     getX: Int,
#     setX: Int->Rep,
#     bump: Rep,
#     setC: Color->Rep,
#     getC: Color
#   end;
CPointM = Fun(Rep)
            {|getX: Rep->Int,  setX: Rep->Int->Rep,  bump: Rep->Rep,
              setC: Rep->Color->Rep,  getC: Rep->Color|}
CPoint = Object CPointM
CPoint'getX : All(M<CPointM) (Object M) -> Int
CPoint'setX : All(M<CPointM) (Object M) -> Int -> (Object M)
CPoint'bump : All(M<CPointM) (Object M) -> (Object M)
CPoint'setC : All(M<CPointM) (Object M) -> Color -> (Object M)
CPoint'getC : All(M<CPointM) (Object M) -> Color
```

5 Inheritance

In the following sections we demonstrate how inheritance can be implemented within the formal framework we have developed so far. It is important to note that the basic theoretical work of the paper is completely finished at this point. We began with a simple model of objects, using existential types to capture the essential notion of encapsulation. This model was refined by the introduction of the `Object` type constructor (moving us from the second-order polymorphic λ-calculus, System F, to a higher-order calculus with a richer set of kinds, System F^{ω}). The introduction of subtyping required another extension of the basic calculus, and the interaction of subtyping and encapsulation forced a crucial refinement in the typing of message-sending functions like `Point'bump`. From this point on, however, we will require *no* further changes, either to our encoding of objects or to the underlying system of types. (At the level of values, on the other hand, we will need to use the fixed-point constructor `rec`, which has not been necessary up until now, to model the behavior of `self`.)

The word "inheritance" is used to describe a variety of language features that allow object definitions to be constructed incrementally by sharing implementations of methods in hierarchies of *classes*. We can think of classes as templates which can either be used to create objects or extended to create new classes. In adopting this definition we also make an important distinction between objects and classes: objects may only be manipulated by sending them messages, their methods may not be extended or changed; classes, on the other hand, may be extended but cannot be sent messages.

In the high-level syntax provided by our prototype compiler, the following declaration creates both a class (a value named `pointClass` whose type, `Class PointM`, may be read as "A class whose instances are objects with interface `PointM`") and, for convenience, an initial instance of this class named `point'new`. (The definition of the type constructor `Class` is given in the next section.)

```
# class point : Point =
#   vars x : Int = 0
#   with
#     setX  =  fun(i:Int) state@x = i,
#     getX  =  state@x,
#     bump  =  state@x = plus state@x 1
#   end;
pointClass : Class PointM
point'new : Object PointM
```

The phrase "`vars x : Int = 0`" declares the internal state type to be a record with a single field `x` of type `Int`, whose initial value in `point'new` is `0`. (We allow more than one internal state variable. For example, the declaration

```
vars x : Int = 0, y : Int = 0
```

introduces two instance variables `x` and `y`, and declares the internal state type to be a record containing two fields `x` and `y` of type `Int`.) The methods `setX`, `getX` and `bump` are defined in terms of an implicit parameter `state`, which represents the internal state of the object. The field `l` of a state `s` is accessed and updated by writing `s@l` and `s@l=i`, respectively. (We shall see later that these do not mean quite the same thing as `s.l` and `{l=i}`, although the intuition is similar.)

Multiple instance variables can be updated by cascaded applications of `@`, as in `(s@l=i)@m=j`. This illustrates why it is necessary to specify *which* state is to be updated by `@`: if we assumed that the first argument of `@` would always be `state`, then there would be no way to update more than one instance variable in the present purely functional framework. In a richer language with side effects, we could omit this argument.

Now, we wish the setX, getX and bump methods of colored points to behave just like those of points. The basic idea of inheritance is to provide a notation that allows these methods to be written just once, in the definition of points, and then reused in the definition of colored points.

```
# class cpoint : CPoint from point : Point =
#   vars c : Color = red
#   inherit setX, getX, bump
#   with
#     setC = fun(c:Color) state@c = c,
#     getC = state@c
#   end;
cpointClass : Class CPointM
cpoint'new : Object CPointM
```

The phrase "from point : Point" in the class header and the inherit clause two lines below indicate that this declaration is not free-standing, but rather defines the behavior of colored points incrementally, with respect to the existing class point. Only the new methods setC and getC are defined explicitly; the other three are taken from point. (In most object-oriented languages, the inherit clause is implicit: all methods not explicitly overridden in a subclass definition are inherited from the superclass. However, compiling such definitions into our low-level typed λ-calculus becomes a little less straightforward.)

Most object-oriented languages carry the idea of inheritance two significant steps further. First, we may wish that the bump method in the definition of points could be implemented in terms of calls to the setX and getX methods, instead of changing the state directly. This is good programming practice, since it localizes the behavior of "setting the x-coordinate" in exactly one definition: the setX method. In typical object-oriented languages, this need is satisfied by providing the ability to send messages to "self" — that is, to the very object executing the method in which self is mentioned. Here, we use a slightly different syntax, viewing self as just a record of methods rather than as a whole object:

```
# class point : Point =
#   vars x : Int = 0
#   with
#     setX = fun(i:Int) state@x=i,
#     getX = state@x,
#     bump = self.setX state (plus (self.getX state) 1)
#   end;
pointClass : Class PointM
point'new : Object PointM
```

This implementation of bump uses the getX method of self to extract the current x coordinate, increments it by 1, and uses self's setX method to store the updated value in the state, yielding a new state, which it returns as its own result. As before, it is necessary to pass the state argument explicitly, since we might want to apply several methods in turn (as in the following example).

Now, imagine that the colored point class provides a new implementation of setX (one that updates the x-coordinate as usual but also changes the point's color to blue, for example):

```
# class cpoint : CPoint from point : Point =
#   vars c : Color = red
#   inherit getX, bump
#   with
#     setX = fun(i:Int)
#              let state' = super.setX state i
#              in self.setC state' blue end,
#     setC = fun(c:Color) state@c=c,
#     getC = state@c
#   end;
cpointClass : Class CPointM
cpoint'new : Object CPointM
```

As in the implementation of bump, we can use self.setC to change the color, rather than modifying it directly. But since we are defining setX, we clearly cannot use self.setX. Nevertheless, since we have already defined the x-coordinate-setting behavior of setX once, it would be ideal if we were not forced to redefine this aspect of the behavior of the new setX, but could refer to the *original* behavior of the setX method of points. This ability is provided by the implicit parameter super in the second line of the definition of the setX method. Also note that, since we are working in a purely functional language, we are forced to be explicit about *which* state is being updated or queried. In the implementation of setX, state is the original state passed to the method as an implicit parameter and state' is the state after the x field has been changed. The final result of the method thus has new values for both x and c.

The second major step taken in many object-oriented language designs is to arrange that the behavior of this new setX is also seen by the bump method (which was defined *earlier* in the class point and inherited by cpoint), so that sending bump to a colored point changes its color to blue as well as incrementing its x-coordinate:

```
# c1 = cpoint'new;
c1 = <val> : Object CPointM
# CPoint'getC CPointM c1;
red : Color
# c2 = Point'bump CPointM c1;
c2 = <val> : Object CPointM
# CPoint'getC CPointM c2;
blue : Color
```

This so-called *late binding* of recursive references in bump to self.setX and self.getX is often cited as a characteristic feature of object-oriented languages. Although we shall see that it is by no means a necessary feature — somewhat simpler and perhaps equally useful variants of inheritance can be built without it — the task of providing it is an interesting challenge, and the fact that it can be provided quite straightforwardly stands as additional evidence that our type theory is rich enough to capture a wide variety of object-oriented features.

6 Implementing Inheritance

We now explain in detail how inheritance can be implemented. We begin with a simple version of inheritance where the instance variables of a class are accessible from methods in its subclasses, working in several stages so as to introduce the more difficult technical constructions one by one. Section 7 develops a more sophisticated implementation where superclass instance variables are hidden from subclasses. (It is this version that our compiler uses as the base for the high-level class definitions described in the previous section.)

From this point through Section 9, the development becomes somewhat more technical. Since the details of inheritance have no bearing on the basic object model developed in the early sections of the paper, some readers may want to skim these sections, or even skip directly to Section 10.

The essential differences between a class and an object are threefold:

1. The internals of an object are protected by a hard encapsulation boundary; there is no way to pull them out, make incremental modifications, and replace the packaging. (Languages in which this sort of incremental modification of objects is allowed are called *delegation-based*; their notion of encapsulation is somewhat different from the kind we are considering.)

2. The methods of an object are specialized to work on internal states of one particular type — typically records with a fixed collection of fields. But subclass definitions may add new fields. In order to deal with this flexibility, the methods of a class must be *polymorphic* in the final representation type.

3. The methods of an object are essentially functions from states to states. But to implement the behavior of `self` introduced in the previous section, it is necessary to postpone deciding which record of methods the special name `self` refers to; in a given class, instances of `self` do not necessarily refer to the methods of that class, but perhaps to the methods of some subclass that has not yet been defined. Thus, the methods in a class should be thought of as functions from `self` to functions from states to states.

A *class*, then, is essentially just an object with enough of the packaging left off, and enough decisions about representation and recursive self-reference postponed, that it can still be extended. When the class is instantiated to form an object, the representation and references to `self` are fixed and the methods all become concrete functions:

$$\begin{tikzcd}
\begin{array}{c}\texttt{pointClass}\\ \in \texttt{Class PointM}\end{array} \arrow[r, "\texttt{new}"] \arrow[d, "\texttt{extend}"] & \texttt{p} \in \texttt{Object PointM} \\
\begin{array}{c}\texttt{cpointClass}\\ \in \texttt{Class CPointM}\end{array} \arrow[r, "\texttt{new}"] & \texttt{cp} \in \texttt{Object CPointM}
\end{tikzcd}$$

Note that classes themselves are values, not types: we can write many different classes of type `Class PointM`, each of which can be used to build objects of type `Object PointM`. Moreover, the type of a colored point class is not a subtype of the type of a point class, although the type of colored point objects is a subtype of the type of point objects.

Let us assume, for the moment, that points and colored points have exactly the same representation type, so that we only need to deal with inheritance of methods:

```
# CommonRep = {| x:Int,color:Color |};
CommonRep : *
```

Then the (second) point class from the previous section can be implemented in pure $F^{\omega}_{\leq}$ as a record of methods, abstracted on a record `self` of methods with the same types:

```
# pointClass =
#   fun(self:PointM CommonRep)
#     {getX = fun(s:CommonRep) s.x,
#      setX = fun(s:CommonRep) fun(i:Int) {x=i,color=s.color},
#      bump = fun(s:CommonRep) self.setX s (plus (self.getX s) 1)}
#     : PointM CommonRep;
pointClass = <val> : (PointM CommonRep) -> (PointM CommonRep)
```

(The type assertion "`: PointM CommonRep`" is included to help the typechecker print the type of `pointClass` in a readable form; without the assertion, an equivalent but more verbose type is printed.)

The recursive references to `setX` and `getX` in `bump` are delayed by referring to the `setX` and `getX` fields from `self`. These delayed references are resolved when we create an instance of `pointClass` by supplying it as the argument to the polymorphic fixed point operator `rec: All(A) (A->A)->A` to create a concrete record of functions, which is then encapsulated as a point object as before:

```
# p = < CommonRep,
#       {state = {x=1,color=red},
#        methods = rec (PointM CommonRep) pointClass}
#     >: Object PointM;
p = <val> : Object PointM
```

The class `pointClass` here has no superclasses; its behavior is defined directly. Colored points, on the other hand, are defined incrementally, by means of a function mapping an implementation of the point methods — called `super` here — to an implementation of the colored point methods:

```
# buildCPointClass =
#   fun(super:PointM CommonRep)      % superclass methods
#   fun(self:CPointM CommonRep)      % recursively defined "self" methods
#     {getX = super.getX,
#      setX = super.setX,
#      getC = fun(s:CommonRep) s.color,
#      setC = fun(s:CommonRep) fun(c:Color) {x=s.x, color=c},
#      bump = super.bump}
#     : CPointM CommonRep;
buildCPointClass = <val>
```

```
                    :    (PointM CommonRep)
                      -> (CPointM CommonRep)
                      -> (CPointM CommonRep)
```

The `getX`, `setX`, and `bump` methods are inherited by copying them from the abstracted record of point methods. The parameter `self` plays the same role here as it did in `pointClass`, delaying recursive references to the colored point methods until instantiation time. (It happens that there are no such references here.)

To create a colored point object, we first use `pointClass` to build an implementation of the inherited point methods, supplying it with a record of colored point methods obtained by taking a fixed point as before (which can be regarded as a record of point methods since `CPointM CommonRep` is a subtype of `PointM CommonRep`):

```
# cp = < CommonRep,
#        {state = {x=1,color=red},
#         methods = rec (CPointM CommonRep)
#                     fun(self: CPointM CommonRep)
#                        buildCPointClass (pointClass self) self}
#      >: Object CPointM;
cp = <val> : Object CPointM
```

Now let us consider the more realistic case where defining a subclass involves extending both the state and the collection of methods. Here, points should be represented using just an `x` field, while the representation of colored points also has a `color` field.

```
# PointR = {| x:Int |};
PointR : *
# CPointR = {| x:Int, color:Color |};
CPointR : *
```

At the same time, we provide general extension and instantiation functions that can be applied to arbitrary class definitions.

The new variability in representations creates a technical difficulty. It is *not* literally true any more that the `setX` method behaves identically in points and colored points: the `setX` of points expects a state argument of type `PointR`, which it discards and replaces with a new value, while the `setX` of colored points expects a state argument of type `CPointR` and returns a record with a new `x` field and a copy of the old `color` field.

To resolve this difficulty, we need the observation that the `setX` method of points does not actually need to know that the state type *is* `PointR`, but only that the state *contains* an x-coordinate. That is, it needs a way of extracting a component of type `PointR` from the state and a way of overwriting just this component to produce a new copy of the state. By abstracting `pointClass` on a pair of functions for extracting (`get`) and overwriting (`put`), we obtain a new point class that is polymorphic in the "final representation type" `FinalR` of some eventual subclass:

```
# pointClass =
#    fun(FinalR)
#    fun(get: FinalR->PointR)
#    fun(put: FinalR->PointR->FinalR)
```

```
#    fun(self: PointM FinalR)
#      {getX = fun(s:FinalR) (get s).x,
#       setX = fun(s:FinalR) fun(i:Int) put s {x=i},
#       bump = fun(s:FinalR) put s {x=(plus (get s).x 1)}
#      }: PointM FinalR;
pointClass = <val>
           : All(FinalR)
                 (FinalR->PointR)
              -> (FinalR->PointR->FinalR)
              -> (PointM FinalR)
              -> (PointM FinalR)
```

Abstracting the method interface `PointM` and the local representation type `PointR` in the type of `pointClass` yields a type operator describing the types of arbitrary class definitions.

```
# Class =
#   Fun(SelfM:*->*)
#   Fun(SelfR)
#     All(FinalR)
#     (FinalR->SelfR) ->
#     (FinalR->SelfR->FinalR) ->
#     (SelfM FinalR) ->
#     (SelfM FinalR);
Class : (*->*)->*->*
```

The generic instantiation function `new` takes a class and an initial state and constructs an object using the fixed-point constructor as before, choosing the "final representation" to be the same as the "local representation," and supplying an identity function as the extractor and, as the overwriter, a two-argument function that simply returns its second argument.

```
# new =
#   fun(SelfM:*->*)
#   fun(SelfR)
#   fun(selfClass: Class SelfM SelfR)
#   fun(s: SelfR)
#       <SelfR,
#        {state = s,
#         methods =
#           rec (SelfM SelfR)
#             (fun(self: SelfM SelfR)
#               selfClass SelfR
#                         (fun(s:SelfR) s)
#                         (fun(s:SelfR) fun(s':SelfR) s')
#                         self)
#       }>: Object SelfM;
new = <val>
    : All(SelfM:*->*)
      All(SelfR)
        (Class SelfM SelfR) -> SelfR -> (Object SelfM)
```

Point objects are created by applying `new` to the type of the point methods, a representation type, an appropriately typed point class, and an initial value of the representation type.

```
# p = new PointM PointR pointClass {x=1};
p = <val> : Object PointM
```

Finally, we can write a generic class extension function, `extend`. This function is abstracted on

- an existing class definition `superClass`,
- a function `inc` that describes the "increment" between the methods of the given class and those of the new class, and
- an extractor `get` and an overwriter `put` for converting between the representation type `SuperR` of the given class and the desired representation type `NewR` of the new class.

Given these parameters, `extend` constructs a new class in which the extractor and overwriter converting between `FinalR` and `NewR` are composed with the ones that convert between `NewR` and `SuperR`, to enable the superclass methods to access their part of the state.

For convenience we first define the `Increment` type. It is like a class type, except that it is also abstracted on the implementation of its superclass methods.

```
# Increment =
#  Fun(SuperM: *->*)   % superclass interface
#  Fun(NewM:   *->*)   % new class interface
#  Fun(NewR)           % new representation
#   All(FinalR)                      % final representation
#     (FinalR->NewR) ->              % extractor
#     (FinalR->NewR->FinalR) ->      % overwriter
#     (SuperM FinalR) ->             % superclass methods
#     (NewM FinalR) ->               % self methods
#     NewM FinalR;     % ...returning the new methods
Increment : (*->*)->(*->*)->*->*

# extend =
#   fun(SuperM:*->*)                      % superclass interface
#   fun(SuperR)                           % superclass representation
#   fun(NewM < SuperM)                    % new class interface
#   fun(NewR)                             % new class representation
#   fun(superClass: Class SuperM SuperR) % the superclass
#   fun(inc: Increment SuperM NewM NewR) % "increment" function
#   fun(get: NewR->SuperR)                % new->super extractor
#   fun(put: NewR->SuperR->NewR)          % new<-super overwriter
#
#    % Build the extended class...
#    (fun(FinalR)
#     fun(g: FinalR->NewR)
#     fun(p: FinalR->NewR->FinalR)
#     fun(self: NewM FinalR)
#       inc FinalR g p
#         (superClass FinalR
#           (fun(s:FinalR) get(g(s)))
```

```
#           (fun(s:FinalR) fun(s':SuperR) p s (put (g s) s'))
#           self)
#         self)
#    : Class NewM NewR;
extend = <val>
       : All(SuperM:*->*)
         All(SuperR)
         All(NewM<SuperM)
         All(NewR)
              (Class SuperM SuperR)
           -> (Increment SuperM NewM NewR)
           -> (NewR->SuperR)
           -> (NewR->SuperR->NewR)
           -> (Class NewM NewR)
```

The colored point class can now be implemented by extending `pointClass`:

```
# cpointClass =
#   extend PointM PointR CPointM CPointR pointClass
#     (fun(FinalR)
#      fun(get: FinalR->CPointR)
#      fun(put: FinalR->CPointR->FinalR)
#      fun(super: PointM FinalR)
#      fun(self: CPointM FinalR)
#        {getX = super.getX,
#         setX = super.setX,
#         getC = fun(s:FinalR) (get s).color,
#         setC = fun(s:FinalR) fun(c:Color) put s {x=0, color=c},
#         bump = super.bump
#        })
#     (fun(s:CPointR) {x=s.x})
#     (fun(s:CPointR) fun(s':PointR) {x=s'.x, color=s.color});
cpointClass = <val> : Class CPointM CPointR
```

Applying `new` to `cpointClass` yields an object of type `Object CPointM`,

```
# cp = new CPointM CPointR cpointClass {x=1, color=red};
cp = <val> : Object CPointM
```

which can be manipulated by sending it messages as in Section 3:

```
# Point'getX CPointM (Point'bump CPointM cp);
2 : Int
```

7 Private Instance Variables

In the literature on object-oriented programming, it has sometimes been argued (e.g. (Snyder, 1986)) that giving subclasses direct access to the instance variables of their superclasses is a violation of proper encapsulation discipline. In this section we develop an alternative implementation of `extend` and `new` where instance variables are hidden from subclasses. Methods defined in the point class will see the same representation

```
# PointR = {| x:Int |};
PointR : *
```

as before, but the new methods defined in the colored point class will only see the new instance variable `color`:

```
# CPointR = {| color:Color |};
CPointR : *
```

We begin by introducing some higher-level operations for manipulating maps between state vectors of different shapes. An "extractor" from a larger type `S` to a smaller type `T`, written `Extractor S T`, is a pair of functions — one for extracting the `T` component of an element of `S` and one for overwriting the `T` component of an element of `S` with a new value:

```
# Extractor = Fun(S) Fun(T) {| get: S->T, put: S->T->S |};
Extractor : *->*->*
```

The simplest extractor is the one that maps between a type `S` and `S` itself:

```
# idExtractor =
#   fun(S)
#     {get = fun(s:S) s,
#      put = fun(s:S) fun(t:S) t}
#     : Extractor S S;
idExtractor = <val> : All(S) Extractor S S
```

Given extractors `e1` and `e2` of appropriate types, we can form the "composition" of `e1` and `e2` as follows:

```
# composeExtractors =
#   fun(T1) fun(T2) fun(T3)
#     fun(e1: Extractor T1 T2)
#       fun(e2: Extractor T2 T3)
#         {get = fun(t1:T1) e2.get (e1.get t1),
#          put = fun(t1:T1) fun(t3:T3)
#                  e1.put t1 (e2.put (e1.get t1) t3)}
#         : Extractor T1 T3;
composeExtractors = <val>
                  : All(T1)
                    All(T2)
                    All(T3)
                         (Extractor T1 T2)
                      -> (Extractor T2 T3)
                      -> (Extractor T1 T3)
```

Finally, we can define extractors for the special case when the larger type `S` is just a pair of the smaller type `T` with some other type:

```
# Pair = Fun(T1) Fun(T2) {| fst:T1, snd:T2 |};
Pair : *->*->*

# fstExtractor =
#  fun(T1) fun(T2)
#    {get = fun(p: Pair T1 T2) p.fst,
```

```
#     put = fun(p: Pair T1 T2) fun(t:T1) {fst=t, snd=p.snd}}
#    : Extractor (Pair T1 T2) T1;
fstExtractor = <val> : All(T1) All(T2) Extractor (Pair T1 T2) T1
```

```
# sndExtractor =
#  fun(T1) fun(T2)
#    {get = fun(p: Pair T1 T2) p.snd,
#     put = fun(p: Pair T1 T2) fun(t:T2) {fst=p.fst, snd=t}}
#    : Extractor (Pair T1 T2) T2;
sndExtractor = <val> : All(T1) All(T2) Extractor (Pair T1 T2) T2
```

The crucial change is in the definition of the operator `Class`. Instead of including an explicit representation type in the type of a class, we existentially quantify the class with respect to a type variable that stands for some hidden representation type that was chosen when the class was built. The initial value of the local state component is also specified in the class, instead of being chosen at instantiation time.

```
# Class =
#   Fun(SelfM:*->*)
#     Some(SelfR)
#       {| localstate: SelfR,
#         buildM: All(FinalR)
#                  (Extractor FinalR SelfR) ->
#                  (SelfM FinalR) ->
#                  (SelfM FinalR) |};
Class : (*->*)->*
```

For example, the class of point objects is:

```
# pointClass =
#   < PointR,
#     {localstate = {x=1},
#      buildM =
#        fun(FinalR)
#        fun(e: Extractor FinalR PointR)
#        fun(self: PointM FinalR)
#          {getX = fun(s:FinalR) (e.get s).x,
#           setX = fun(s:FinalR) fun(i:Int) e.put s {x=i},
#           bump = fun(s:FinalR) self.setX s (plus (self.getX s) 1)}}
#   >: Class PointM;
pointClass = <val> : Class PointM
```

The `new` function obtains the representation type for the new object by opening the class to reveal its hidden representation type and initial state; since these are immediately used to create a new object that also hides its representation, the representation type is not allowed to escape.

```
# new =
#   fun(SelfM:*->*)
#   fun(selfClass: Class SelfM)
#     open selfClass as <SelfR,selfData> in
#       < SelfR,
#         {state = selfData.localstate,
```

```
#          methods =
#            rec (SelfM SelfR)
#              (fun(self:SelfM SelfR)
#                selfData.buildM SelfR (idExtractor SelfR) self)}
#        >: Object SelfM
#      end;
new = <val> : All(SelfM:*->*) (Class SelfM) -> (Object SelfM)
```

Similarly, the `extend` function opens the packaged superclass to reveal its representation type and initial state, and uses these to form the representation type and initial state of the new class by pairing them with the new local representation type `NewDeltaR` and the new local state `deltastate`. Extractors for the local state of the superclass (needed by the function `superData.buildM`, which builds the superclass methods) and the new class (needed by the function `build`, which builds the new local methods) are constructed from an extractor from the final representation type to the new state type — the pair of the new local state type and the superclass state type — by composing it with `fstExtractor` and `sndExtractor`.

```
# extend =
#    fun(SuperM:*->*)
#    fun(NewM < SuperM)
#    fun(NewDeltaR)
#    fun(superClass: Class SuperM)
#    fun(deltastate: NewDeltaR)
#    fun(inc: Increment SuperM NewM NewDeltaR)
#      open superClass as <SuperR,superData> in
#       < Pair SuperR NewDeltaR,
#         {localstate = {fst = superData.localstate,
#                        snd = deltastate},
#          buildM =
#            fun(FinalR)
#            fun(e: Extractor FinalR (Pair SuperR NewDeltaR))
#            fun(self: NewM FinalR)
#             let esnd = composeExtractors
#                          FinalR (Pair SuperR NewDeltaR) (NewDeltaR)
#                          e (sndExtractor SuperR NewDeltaR)
#             in inc FinalR esnd.get esnd.put
#               (superData.buildM FinalR
#                  (composeExtractors FinalR (Pair SuperR NewDeltaR) SuperR e
#                                     (fstExtractor SuperR NewDeltaR))
#                  self)
#               self
#             end}
#        >: Class NewM
#       end;
extend = <val>
       : All(SuperM:*->*)
         All(NewM<SuperM)
         All(NewDeltaR)
              (Class SuperM)
           -> NewDeltaR
           -> (Increment SuperM NewM NewDeltaR)
```

```
        -> (Class NewM)
```

The class of colored points is now defined by extending the point class:

```
# cpointClass =
#   extend PointM CPointM CPointR pointClass
#     {color=red}
#     (fun(FinalR)
#      fun(get: FinalR->CPointR)
#      fun(put: FinalR->CPointR->FinalR)
#      fun(super: PointM FinalR)
#      fun(self: CPointM FinalR)
#        {getX = super.getX,
#         setX = super.setX,
#         getC = fun(s:FinalR) (get s).color,
#         setC = fun(s:FinalR) fun(c:Color) put s {color=c},
#         bump = super.bump
#        });
cpointClass = <val> : Class CPointM

# cp = new CPointM cpointClass;
cp = <val> : Object CPointM
# Point'getX CPointM (Point'bump CPointM cp);
2 : Int
```

The high-level class definitions of Section 5 can be compiled very straightforwardly into calls to the `extend` function defined in this section. Method bodies are implicitly abstracted on a state vector `state`, so that the `setX` method

```
fun(i:Int) state@x=i
```

becomes:

```
fun(state:{x:Int}) fun(i:Int) put state {x=i}
```

The notation for accessing and updating instance variables — `s@li` for accessing the field with the $\mathtt{i}^{th}$ label and `s@li=e` for updating the `li` field of the state vector `s` — is compiled into calls to `get` and `put`:

```
s@li   =  (get s).li

s@li=e  =  put s {l1=(get s).l1, ..., li=e, ..., ln=(get s).ln}
```

where `{l1..ln}` is the set of instance variables in the local part of the state vector. (Although the latter abbreviation is a kind of polymorphic record update, it can always be translated directly into lower-level record operations, since the set of instance variable names in a given class definition is always known statically; the powerful extensible record types of (Wand, 1987; Rémy, 1989; Cardelli, 1992a; Cardelli & Mitchell, 1991; Jategaonkar & Mitchell, 1988) are not required.)

Of course, in a setting where instance variables were mutable, references to the implicit state variable `state` could be dropped, since there would never be any need to apply `get` or `put` to any state vector other than `state`.

8 Example: Smalltalk-Style Collections

The Smalltalk collection classes are often cited as a paradigm example of the use of inheritance in object-oriented programming. The standard Smalltalk-80 programming environment includes a rich variety of class definitions for data structures representing sets, bags, lists, arrays, and other sorts of collections. The definitions of these classes are organized in an inheritance hierarchy so that a great deal of functionality is shared between groups of behaviorally similar classes. In this section, we implement a simple, purely functional variant of the classes `Collection` and `Bag`.

For the moment, we assume that the elements of a collection are always integers. The class `IntCollection`, which forms the root of the hierarchy of collection classes, describes the behavior common to all integer collections. We provide just two operations: `size`, which counts the elements of a collection, and `fold`, which applies a given function to all the elements of a collection in turn, passing the result of the previous application as the second argument in each case:

```
# IntCollection =
#   ObjectType(Rep) with
#      fold: All(A) (Int->A->A) -> A -> A,
#      size: Int
#   end;
IntCollectionM = Fun(Rep)
                   {|fold: Rep->(All(A)(Int->A->A)->A->A),  size: Rep->Int|}
IntCollection = Object IntCollectionM
IntCollection'fold : All(M<IntCollectionM)
                       (Object M) -> (All(A)(Int->A->A)->A->A)
IntCollection'size : All(M<IntCollectionM) (Object M) -> Int
```

The `size` method can be implemented straightforwardly in terms of `fold`. But `fold` itself cannot be implemented generically for an arbitrary collection; its behavior depends on the specific sort of collection in question; in other words, `fold` is a *virtual* (or *deferred*) method, which must be supplied in the subclasses of `IntCollection`:

```
# class intCollection : IntCollection =
#   virtual fold
#   with
#     size = self.fold state Int
#              (fun(elt:Int) fun(count:Int) succ count)
#              0
#   end;
intCollectionClass : Class IntCollectionM
intCollection'new : Object IntCollectionM
```

The keyword `virtual` is like `inherits`, except that it directs the compiler to copy the listed methods from the `self` method vector rather than from `super`. Its translation into pure $F^{\omega}_{\leq}$ is given in the next section.

A bag is a simple sort of concrete collection. Here, we provide just one operation in addition to `fold` and `size`: an `add` method that inserts a new element into a bag:

```
# IntBag =
#   ObjectType(Rep) with
#      fold: All(A) (Int->A->A) -> A -> A,
#      size: Int,
#      add:  Int -> Rep
#   end;
IntBagM = Fun(Rep)
            {|fold: Rep->(All(A)(Int->A->A)->A->A),  size: Rep->Int,
              add: Rep->Int->Rep|}
IntBag = Object IntBagM
IntBag'fold : All(M<IntBagM) (Object M) -> (All(A)(Int->A->A)->A->A)
IntBag'size : All(M<IntBagM) (Object M) -> Int
IntBag'add : All(M<IntBagM) (Object M) -> Int -> (Object M)
```

The class declaration for bags must implement `add` (since it is new) and `fold` (since it was declared as virtual), but it can inherit `size` from the superclass. We use a list of integers to represent the elements of a bag and the `foldList` function to fold a function over the list representing the elements of a bag:

```
# class intBag : IntBag from intCollection : IntCollection =
#   vars l : List Int = nil Int
#   inherit size
#   with
#     fold = fun(A)
#            fun(f: Int->A->A)
#            fun(a: A)
#              foldList Int state@l A f a,
#     add  = fun(i:Int)
#              state@l = (cons Int i state@l)
#   end;
intBagClass : Class IntBagM
intBag'new : Object IntBagM
```

The function `cons` here builds a new list from an old list and an element to be added to the front; we can build lists of any type, so we instantiate `cons` by naming the type of the elements as its first argument.

Note that the definition of `size` that `intBag` inherits from `intCollection` makes an internal call to the `fold` method of `intBag`. This capability is the essence of Smalltalk-style inheritance.

We can now write a simple program that builds a bag and calculates its size:

```
# b1 = IntBag'add IntBagM intBag'new 7;
b1 = <val> : Object IntBagM
# b2 = IntBag'add IntBagM b1 88;
b2 = <val> : Object IntBagM
# IntCollection'size IntBagM b2;
2 : Int
```

9 Polymorphic Collections

Of course, we'd like to be able to build collections with elements of any type whatsoever, not just integers. This can be accomplished by a straightforward generalization of our implementation of inheritance. To simplify the presentation, we return to the variant of inheritance developed in Section 6, where instance variables of superclasses are visible to subclasses.

First, we introduce the notion of a *polymorphic class* — a class, in the sense of Section 6, abstracted on an additional type parameter `E`. Since the concrete representation type depends on the eventual value of `E`, the types `SelfR` and `FinalR` become one-argument type operators; similarly, the interface `SelfM` becomes a two-argument operator (one argument, as before, stands for the hidden representation type; the other stands for the element type).

```
# PolyClass =
#   Fun(SelfM:*->*->*)
#   Fun(SelfR:*->*)
#     All(E)
#     All(FinalR:*->*)
#     (FinalR E->SelfR E) ->
#     (FinalR E->SelfR E->FinalR E) ->
#     (SelfM E (FinalR E)) ->
#     (SelfM E (FinalR E));
PolyClass : (*->*->*)->(*->*)->*
```

For example, since the collection class has no instance variables of its own, its local representation type is expressed by the operator:

```
# CollectionR = Fun(E) {| |};
CollectionR : *->*
```

The interface of the collection methods is:

```
# CollectionM =
#    Fun(E) Fun(Rep)
#      {| fold: Rep -> All(A) (E->A->A) -> A -> A,
#         size: Rep -> Int |};
CollectionM : *->*->*
```

The collection class itself is formed from the class of integer collections by abstracting on `E`:

```
# collectionClass =
#  (fun(E)
#   fun(FinalR:*->*)
#   fun(get: FinalR E->CollectionR E)
#   fun(put: FinalR E->CollectionR E->FinalR E)
#   fun(self: CollectionM E (FinalR E))
#     {fold = self.fold,
#      size = fun(state:FinalR E)
#               self.fold state Int
#               (fun(elt:E) fun(count:Int) succ count)
#               0})
#     : PolyClass CollectionM CollectionR;
collectionClass = <val> : PolyClass CollectionM CollectionR
```

The functions `polynew` and `polyextend` are the evident generalizations of `new` and `extend`. We show just `polyextend` here:

```
# PolyIncrement =
#  Fun(SuperM: *->*->*)   % superclass interface
#  Fun(NewM:   *->*->*)   % new class interface
#  Fun(NewR: *->*)        % new representation
#    All(E)                              % element type
#    All(FinalR:*->*)                    % final representation
#    (FinalR E->NewR E) ->               % extractor
#    (FinalR E->NewR E->FinalR E) ->     % overwriter
#    (SuperM E (FinalR E)) ->            % superclass methods
#    (NewM E (FinalR E)) ->              % self methods
#    (NewM E (FinalR E));           % ...returning the new methods
PolyIncrement : (*->*->*)->(*->*->*)->(*->*)->*
# polyextend =
#   fun(SuperM:*->*->*)                        % superclass interface
#   fun(SuperR:*->*)                           % superclass representation
#   fun(NewM < SuperM)                         % new class interface
#   fun(NewR:*->*)                             % new class representation
#   fun(superClass: PolyClass SuperM SuperR)    % the superclass
#   fun(build: PolyIncrement SuperM NewM NewR)  % "increment" function
#   fun(get: All(E) NewR E->SuperR E)          % new->super extractor
#   fun(put: All(E) NewR E->SuperR E->NewR E)   % new<-super overwriter
#
#    % Build the extended class...
#    (fun(E)
#     fun(FinalR:*->*)
#     fun(g: FinalR E->NewR E)
#     fun(p: FinalR E->NewR E->FinalR E)
#     fun(self: NewM E (FinalR E))
#       build E FinalR g p
#         (superClass E FinalR
#           (fun(s:FinalR E) get E (g(s)))
#           (fun(s:FinalR E) fun(s':SuperR E) p s (put E (g s) s'))
#           self)
#         self)
#    : PolyClass NewM NewR;
polyextend = <val>
           : All(SuperM:*->*->*)
             All(SuperR:*->*)
             All(NewM<SuperM)
             All(NewR:*->*)
                  (PolyClass SuperM SuperR)
               -> (PolyIncrement SuperM NewM NewR)
               -> (All(E)(NewR E)->(SuperR E))
               -> (All(E)(NewR E)->(SuperR E)->(NewR E))
               -> (PolyClass NewM NewR)
```

Now the representation and interface specification of polymorphic bags are

```
# BagM =
#    Fun(E) Fun(Rep)
#      {| fold: Rep -> All(A) (E->A->A) -> A -> A,
#         size: Rep -> Int,
#         add:  Rep -> E -> Rep |};
BagM : *->*->*
```

```
# BagR = Fun(E) {| elements : List E |};
BagR : *->*
```

and a suitable class definition is:

```
# bagClass =
#   polyextend CollectionM CollectionR BagM BagR collectionClass
#   (fun(E)
#    fun(FinalR:*->*)
#    fun(g: FinalR E->BagR E)
#    fun(p: FinalR E->BagR E->FinalR E)
#    fun(super: CollectionM E (FinalR E))
#    fun(self: BagM E (FinalR E))
#      {add = fun(state:FinalR E)
#               fun(newelt:E)
#                 p state {elements = cons E newelt (g state).elements},
#       fold = fun(state:FinalR E)
#                fun(R)
#                  fun(f:E->R->R) fun(start:R)
#                    foldList E (g state).elements R f start,
#       size = super.size})
#    (fun(E) fun(s:BagR E) {})
#    (fun(E) fun(s:BagR E) fun(s':CollectionR E) s);
bagClass = <val> : PolyClass BagM BagR
```

When we create a bag, we choose both the type of its elements and the initial value of its internal state:

```
# mybag = polynew BagM Color BagR
#           bagClass
#           {elements = nil Color};
mybag = <val> : Object (BagM Color)
```

The message-sending functions `Bag'add`, `Collection'size`, etc. are also abstracted on the type of the elements:

```
# Bag'add;
<val> : All(E) All(M<BagM E) (Object M) -> E -> (Object M)
# Collection'size;
<val> : All(E) All(M<CollectionM E) (Object M) -> Int
```

We can send messages to the integer bag `mybag` as follows:

```
# mybag1 = Bag'add Color (BagM Color) mybag blue;
mybag1 = <val> : Object (BagM Color)
# mybag2 = Bag'add Color (BagM Color) mybag1 red;
mybag2 = <val> : Object (BagM Color)
# Collection'size Color (BagM Color) mybag2;
2 : Int
```

10 Related Work

Bruce (1992; 1993) develops a formal semantics (based on previous models by Mitchell (1990), Cook (1990), and their collaborators) and a proof of soundness for a high-level object-oriented language with essentially the same features as ours. Bruce's model is fundamentally quite similar to the one developed here. In particular, his use of F-bounded quantification corresponds to our use of higher-order bounded quantification, and his *inh* relation between object types corresponds to operator subtyping. The principal difference is that Bruce uses recursive types instead of type operators to represent the interface types of objects, leading him to conclude that:

> "While the semantics of our language is rather complex, involving fixed points at both the element and type level, we believe that this complexity underlies the basic concepts of object-oriented programming languages. Inherently complex features include the implicit recursion inherent in the keyword, *self*, to refer to the current object, and its corresponding type..." (Bruce, 1992, abstract)

While we agree that fixed points at the element level are required to model the inheritance of methods referring to *self*, we have argued that the complexity of recursive types is *not* inherent in the basic concepts of object-oriented programming. Bruce's account also seems to be complicated by the fact that it uses both recursive types (for interfaces) and existential types (for hiding instance variables); it is not clear why both should be needed.

We formulate our account in terms of more primitive record operations than Bruce, using explicit extractors and overwriters to handle extension of the state during inheritance; Bruce uses extensible records (Rémy, 1989; Cardelli & Mitchell, 1991) for this purpose. Of course, our translation from the high-level syntax described in Section 5 into pure $F^{\omega}_{\leq}$ must generate appropriate extractors and overwriters, which amounts to implementing a kind of extensible records; however, since the set of fields of a record being extended is always known statically, the full complexity of row variables (Wand, 1987; Cardelli, 1992a) is not needed.

Cardelli's treatment of object-oriented programming (1992a; 1992b) aims to describe the same basic features of encapsulation, subtyping, and inheritance as Bruce's and ours. Like us (and unlike Bruce), Cardelli adopts a syntactic point of view, trying to capture a set of fundamental requirements in the form of a typed λ-calculus. Like Bruce (and unlike us), his basic model of objects is recursive records; consequently, recursive record types are used in a critical way. Instead of F-bounded quantification, however, Cardelli uses a set of flexible record extension operators based on the concept of *rows* to achieve the degree of abstraction necessary to support inheritance. Finally, Cardelli shares our concern with formal economy: his high-level calculus of extensible records can be faithfully translated into a pure calculus of bounded quantification. In one dimension, this low-level calculus is simpler than ours: it uses only second-order bounded quantification, while we require higher-order bounded quantification. On the other hand, ours is simpler in that it omits recursive types in favor of existential types (which can themselves be encoded using only universal quantification).

Abadi (1993) and Mitchell, Honsell, and Fisher (1993) present related models of objects and *delegation-based* inheritance (Ungar & Smith, 1987). In both of these systems, a basic λ-calculus-like formalism is extended with new syntactic forms designed to directly capture the operations of message-sending and object construction. A semantics of the extended language is given, and a set of typing rules (implicitly based on recursive types) is proved sound with respect to the semantics.

The difference between class-based formulations of objects and inheritance and formulations based on delegation appears to be mainly one of style and notation: the same formal problems arise in both cases, and they can be handled by similar techniques. Indeed, it is not hard to modify our encodings of objects and message-sending from Sections 2 and 3 to obtain a simple, statically typed model of delegation.

A more significant difference between all these models and the one proposed here arises from the fact that we use existential quantification rather than recursive types types to capture the notion of encapsulation. This foundational difference gives rise to a slight difference in expressive power, which appears in the treatment of binary (in general, n-ary) methods — methods whose list of arguments includes objects of the very same type as the receiver. Such methods can be divided into two essentially different categories:

1. *Strong binary methods*, whose implementation depends on the ability to obtain direct, concrete access to the internal states of several objects at the same time. The typical example of a strong binary method is a `union` operation on sets of integers, where the internal representation of sets of integers is some efficient data structure such as a balanced tree. The internal representation is not normally exposed in the interface of set objects, but the implementation of `union` must be able to obtain the internal representations of *both* arguments in order to perform its task with acceptable efficiency.

2. *Weak binary methods*, which accept one or more arguments of the same type as the receiver, but which need not access the internal states of these extra arguments directly. The usual example discussed in the literature on static type systems for objects, equality methods for point objects, falls into this category; since all the important components of a point's internal state are exposed in its public interface, it is possible to compare one point to another by looking directly at the x coordinate of one of them and comparing it to the number obtained by *asking* the other for its x coordinate.

Models of objects based on recursive types support the use of weak binary methods but not strong ones. Our model supports neither directly. However, in (Pierce & Turner, 1993b), we propose an easy generalization of the basic object model, based on Cardelli and Wegner's *partially abstract types* (1985), that supports the strong form of binary methods. Indeed, since this generalization is based on type-theoretic machinery already available in $F^{\omega}_{\leq}$, the ideas apply equally to any model based on higher-order subtyping.

A more abstract characterization of object types, studied by Hofmann and Pierce (1994), can be used to relate encodings based on existential types and those based on recursive types by showing that both can be viewed as valid implementations for a type system with a primitive *Object* type constructor.

Recent papers by Castagna, Ghelli, and Longo (Castagna *et al.*, 1992; Castagna, 1992; Ghelli, 1991) have proposed an intriguing new approach to the foundations of object-oriented programming. Taking overloading and subtyping as basic, rather than encapsulation and subtyping, they develop an underlying calculus that promises to model some features — notably the *multi-methods* of languages such as CLOS (Bobrow *et al.*, 1988) — that fall completely outside the scope of previous theories, including ours. Indeed, one of the benefits of their work is that, by comparing it to other type-theoretic models of objects, one sees very clearly how essentially different are the basic premises of object-oriented languages such as Simula and Smalltalk, where messages have exactly one receiver and a strong notion of encapsulation is maintained, from languages in the family of CLOS, which give up the strong notion of encapsulation in return for a more symmetric notion of method-body selection based on the types of any number of arguments. The magnitude of this difference is underscored by the fact that modeling CLOS-like multi-methods would seem to require a formal language with some non-parametric notion of *run-time* computation on types.

Most existing object-oriented languages include mutable instance variables. Mutable state can also be provided in our framework by extending $F^{\omega}_{\leq}$ with a *Ref* type constructor similar to that found in ML and specifying a call-by-value reduction strategy. This necessitates a small change in the fixed-point operator used during object creation, but our basic object model is unaffected. Bruce and van Gent (Bruce & van Gent, 1993) describe a similar extension of Bruce's TOOPL language (Bruce, 1993).

Our approach can also be extended to a typed account of *multiple* inheritance by adding intersection types (Coppo *et al.*, 1981) to $F^{\omega}_{\leq}$ (Compagnoni & Pierce, 1993).

Acknowledgements

Our ideas have been sharpened by conversations with Dave Berry, Kim Bruce, Luca Cardelli, Giuseppe Castagna, William Cook, Giorgio Ghelli, Carl Gunter, Bob Harper, Giuseppe Longo, Dave MacQueen, Robin Milner, Kevin Mitchell, Randy Pollack, Nick Rothwell, Stuart Schieber, and Phil Wadler. We are especially grateful to Martin Hofmann for important insights into the formal background of our constructions and to Randy Pollack for helping us understand $F^{\omega}_{\leq}$. Kim Bruce, Adriana Compagnoni, Mike Fourman, Bob Harper, Martin Hofmann, and Kevin Mitchell suggested a number of improvements to earlier drafts. Comments from the referees helped us clarify many details and suggested some important large-scale reorganization of our presentation.

An earlier version of this paper was presented at POPL '93 (Pierce & Turner, 1993a). This work was jointly supported by Harlequin Limited, the U.K. Science and Engineering Research Council, and the ESPRIT Basic Research Actions TYPES and CONFER.

A Introduction to $F^{\omega}_{\leq}$

This appendix gives a short review of $F^{\omega}_{\leq}$, the explicitly-typed λ-calculus used throughout the paper as the formal basis of our encoding of objects. Formally, the type system is a straightforward generalization of Cardelli and Wegner's bounded quantification (1985) with a notion of type operator familiar from Girard's system F^{ω} (1972). The syntax and typing rules are summarized in Appendix B.

The examples in the paper were typeset by a prototype compiler for $F^{\omega}_{\leq}$ that typechecks and evaluates declarations preceded by the symbol `#`. Declarations may be split across a number of lines, and are terminated with a semicolon. A `%` symbol indicates that the rest of the line is a comment.

The compiler's response to an expression is to print its value and type (complex values are printed as `<val>`):

```
# 1;
1 : Int
```

Variables always begin with a lowercase letter; this allows us to distinguish variables from type variables, which start with uppercase letters. We bind top-level expressions to variables by writing `id = e`. For example:

```
# five = 5;
five = 5 : Int
```

Record values are written as `{l = e, ..., l' = e'}`. (Note that record types use slightly different brackets.) We select elements of a record using the syntax `e.x`, where `x` is a label:

```
# record = {x = 1, y = 2};
record = <val> : {|x: Int,  y: Int|}
# record.x;
1 : Int
```

The notion of *subtyping* (Cardelli & Wegner, 1985) formalizes the observation that values of certain types may always be safely substituted for values of other types. For example, we can allow a record of type `{|x: Int, y: Int|}` to be used in a context expecting a record of type `{|x: Int|}`, since presence of the extra field cannot be detected in such a context and so will never lead to run-time error. The subtype relation is defined by a collection of inference rules (listed in Appendix B) with conclusions of the form $\Gamma \vdash S \leq T$.

For example, we use the usual rule (c.f. (Cardelli, 1986)) for subtyping between record types:

$$\frac{\begin{array}{c}\{l_1, \ldots, l_n\} \subseteq \{k_1, \ldots, k_m\} \\ \text{for each } k_i = l_j,\ \ \Gamma \vdash S_i \leq T_j \\ \Gamma \vdash \{\!|k_1{:}S_1, \ldots, k_m{:}S_m|\!\} \in \star\end{array}}{\Gamma \vdash \{\!|k_1{:}S_1, \ldots, k_m{:}S_m|\!\} \leq \{\!|l_1{:}T_1, \ldots, l_n{:}T_n|\!\}} \qquad \text{(S-Record)}$$

Consider, for example, the `extract` function, which extracts the `x`-field from its argument record `r` (we write λ-abstraction using the syntax `fun(x:T)e`):

```
# extract = fun (r: {|x: Int|}) r.x;
extract = <val> : {|x:Int|} -> Int
```

Subtyping allows the `extract` function to accept not only records of type `{|x: Int|}` as arguments, but records of any type which is a subtype of `{|x: Int|}`:

```
# extract {x = 7, y = 8};
7 : Int
```

As usual, the subtyping behavior of the function type constructor is contravariant in the function argument type and covariant in the result type. Intuitively, a function may replace another function if it makes fewer demands on its arguments and gives a better result:

$$\frac{\Gamma \vdash T_1 \leq S_1 \qquad \Gamma \vdash S_2 \leq T_2 \qquad \Gamma \vdash S_1 {\rightarrow} S_2 \in \star}{\Gamma \vdash S_1 {\rightarrow} S_2 \leq T_1 {\rightarrow} T_2} \tag{S-Arrow}$$

We can abstract a type variable `A` from a term `e` using the syntax `fun(A<T) e`. The *bound*, `T`, for the abstracted type variable ensures that any instantiation of `A` will be a subtype of `T`. Thus, we can write the following function, which is polymorphic in the type `A` but requires that `A` is a record type containing an `x` field of type `Int`. (Type application uses the syntax "`e T`".)

```
# extractX = fun (A < {|x: Int|}) fun(r: A) {fst = r.x, snd = r};
extractX = <val> : All(A<{|x:Int|}) A -> {|fst:Int, snd:A|}
# extractX {|x: Int, y: Int|} {x = 5, y = 6};
<val> : {|fst: Int,  snd: {|x:Int, y:Int|}|}
```

The following operations on booleans and integers are built in:

```
false : Bool
true  : Bool
not   : Bool -> Bool
and   : Bool -> Bool -> Bool

plus  : Int -> Int -> Int
minus : Int -> Int -> Int
eqInt : Int -> Int -> Bool
```

Our syntax for existential types is fairly standard. Consider the following implementation of counters based on the representation type `Int`:

```
# counterImpl = {
#   zero = 0,
#   inc = fun(x: Int) plus x 1,
#   isZero = fun(x: Int) eqInt x 0
# };
counterImpl = <val> : {|zero: Int,  inc: Int->Int,  isZero: Int->Bool|}
```

To hide the representation type `Int`, yielding an abstract type of counters, we use the syntax `<T, e> : T'`, where `T` is the actual representation type and `T'` is an existential type that specifies the abstract type's external interface (neither type annotation may be omitted).

```
# counter =
#   % The implementation
#   <Int, counterImpl> :
#   % The interface type
#     Some(C) {|
#       zero : C,
#       inc : C -> C,
#       isZero : C -> Bool
#     |};
counter = <val> : Some(C) {|zero: C,  inc: C->C,  isZero: C->Bool|}
```

Since we have subtyping, we allow a bound for the existentially quantified type variable `C` (this provides what are known as *partially abstract types* (Cardelli & Wegner, 1985)).

Abstract types are unpacked using the `open` construct. In the example below, unpacking `counter` binds the hidden counter representation to `C`, and binds the implementation to `impl`:

```
# open counter as <C,impl> in
#   impl.isZero(impl.inc impl.zero)
# end;
false : Bool
```

The rules for existential types ensure that the only way we can create a counter is to use the operator `impl.zero`, and similarly the only way to modify or examine a counter is by using `impl.inc` and `impl.isZero`.

$F^{\omega}_{\leq}$ incorporates Girard's notion of *type operators* (Girard, 1972), which can be thought of as forming a simply-typed λ-calculus at the level of types. To ensure their well-formedness, types and type operators are assigned *kinds*, `K`, which have the form $*$ or `K->K`. Expressions of kind $*$ are ordinary types; expressions of kind `*->*` are functions from types to types; etc. We can bind types and type operators to type variables in the same way as we did for expressions; the compiler responds to a type or type operator definition by printing its kind.

```
# T = Int -> Int;
T : *
# (fun (x: Int) x) : T;
<val> : T
```

Here we declare the type `T` and use it as a type annotation `:T` on the preceding expression. The typechecker simply checks that the annotated type is equivalent to the type of the expression (type annotations are usually used to simplify the types printed by the typechecker).

Abstraction for type operators uses the syntax `Fun(A:K)`, the uppercase `F` in `Fun` indicating that we are defining a function from types to types rather than from values to values.

```
# Pair = Fun(A: *) Fun(B: *) {| fst: A, snd: B |};
Pair : *->*->*
```

```
# BothBool = Fun(F: *->*->*) F Bool Bool;
BothBool : (*->*->*)->*
# { fst = true, snd = false } : BothBool Pair;
<val> : BothBool Pair
```

(The compiler allows us to omit the kind annotation in the abstraction `Fun(A:K)` whenever `K` is $*$; we could have written `Fun(A) Fun(B) {| fst: A, snd: B |}` in this example.)

Every kind `K` has a maximal element, written `Top(K)`. Our syntax allows the bound of a variable to be omitted if it is `Top(K)`, so that, for example, `Some(C)` actually abbreviates `Some(C<Top(*))`. This type can also be written `Some(C:*)`.

We use pointwise subtyping of operators: `Fun(A:K) T1` is a subtype of `Fun(A:K) T2` if `T1` is a subtype of `T2` under all legal substitutions for `A`. Since `T1` has to be a subtype of `T2` under *all* possible substitutions for `A`, we cannot make any assumptions about `A`; formally, it suffices to check that `T1` is a subtype of `T2` under the assumption that `A < Top(K)`. For example, `Fun(T) {|a:T,b:T|}` is a subtype of `Fun(T) {|a:T|}`, since `{|a:T,b:T|}` is a subtype of `{|a:T|}`.

B 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} (Girard, 1972) with subtyping. The ideas behind this system are due to Cardelli, particularly to his 1988 paper, "Structural Subtyping and the Notion of Power Type" (Cardelli, 1988b); the extension of the subtype relation to type operators was developed by Cardelli and Mitchell (Cardelli, 1990; Mitchell, 1990; Bruce & Mitchell, 1992). Cardelli (Cardelli, 1990) has given a more powerful treatment of operator subtyping, including both monotonic and antimonotonic subtyping in addition to pointwise subtyping.

We omit a detailed treatment of the semantics of $F^{\omega}_{\leq}$. For the examples in this paper, it suffices to regard the meaning of a term as the normal form of its type-erasure (our compiler uses a call-by-name, untyped reduction strategy). A semantic model of a version of $F^{\omega}_{\leq}$ extended with recursive types (and including recursively defined values, which are needed here to model `self`) has been given by Bruce and Mitchell (Bruce & Mitchell, 1992).

B.1 Syntax

B.1.1. Notation: The typing rules that follow define sets of valid judgements of the following forms:

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

$\Gamma \vdash S \sim T$ abbreviates $\Gamma \vdash S \leq T$ and $\Gamma \vdash T \leq S$.

B.1.2. Definition: The sets of kinds, types, terms, and contexts are defined by the following abstract grammar:

K	$::=$	$\star$	kind of types
	$\mid$	$K \rightarrow K$	kind of type operators
T	$::=$	A	type variable
	$\mid$	$\mathrm{Fun}\,(A{:}K)\,T$	type operator
	$\mid$	$T\ T$	application of an operator
	$\mid$	$\mathrm{Top}(K)$	top type
	$\mid$	$T \rightarrow T$	function type
	$\mid$	$\mathrm{All}\,(A{\leq}T)\,T$	universally quantified type
	$\mid$	$\mathrm{Some}\,(A{\leq}T)\,T$	existentially quantified type
	$\mid$	$\{\mid l_1{:}T_1, \ldots, l_n{:}T_n \mid\}$	record type
e	$::=$	x	variable
	$\mid$	$\mathrm{fun}\,(x{:}T)\,e$	abstraction
	$\mid$	$e\ e$	application
	$\mid$	$\mathrm{fun}\,(A{\leq}T)\,e$	type abstraction
	$\mid$	$e\ T$	type application
	$\mid$	$\langle T, e\rangle{:}T$	packing
	$\mid$	$\mathrm{open}\ e\ \mathrm{as}\ \langle A, x\rangle\ \mathrm{in}\ e\ \mathrm{end}$	unpacking
	$\mid$	$\{l_1 = e_1, \ldots, l_n = e_n\}$	record construction
	$\mid$	$e\,.\,l$	field selection
Γ	$::=$	$\bullet$	empty context
	$\mid$	$\Gamma,\ x{:}T$	variable binding
	$\mid$	$\Gamma,\ A{\leq}T$	type var binding with bound

B.1.3. Convention: Whenever we write $\Gamma,\ A{\leq}T$ or $\Gamma,\ x{:}T$ we implicitly require that A and x are not already defined in Γ.

B.1.4. Definition: A type T is *closed* with respect to a context Γ if $FTV(T) \subseteq dom(\Gamma)$. A term e is closed with respect to Γ if $FTV(e) \cup FV(e) \subseteq dom(\Gamma)$. A context Γ is closed if

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

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

B.1.5. 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. Moreover, we assume that all variables bound in a context have distinct names. This convention, which amounts to regarding all variables as bound and viewing bound variables as deBruijn indices (de Bruijn, 1972), replaces the usual side-conditions in rules such as T-Some-E.

B.2 Contexts

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

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

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

B.3 Kinding

The K-TVar rule finds the kind of A by simply looking up the bound associated with A in the context, and then finding the kind of the bound. For example, if the type variable A has been introduced using the K-Arrow-I rule, then the context contains a bound $A \leq \mathrm{Top}(K)$ for some K. However, using the K-Top rule we have that $\mathrm{Top}(K) \in K$ and so, using the K-TVar rule we have that $A \in K$ as expected.

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

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

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

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

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

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

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

$$\frac{\vdash \Gamma \text{ context} \qquad \text{for each } i,\ \Gamma \vdash T_i \in \star}{\Gamma \vdash \{\!|l_1{:}T_1, \ldots, l_n{:}T_n|\!\} \in \star} \qquad \text{(K-Record)}$$

B.4 Subtyping

$$\frac{\Gamma \vdash U \leq S \qquad \Gamma \vdash T \in K \qquad S =_\beta T}{\Gamma \vdash U \leq T} \tag{S-Conv}$$

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

$$\frac{\Gamma \vdash T \in K}{\Gamma \vdash T \leq T} \tag{S-Refl}$$

$$\frac{\Gamma \vdash S \leq T \qquad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U} \tag{S-Trans}$$

$$\frac{\Gamma \vdash S \in K \qquad \Gamma \vdash \text{Top}(K')\, T_1, \ldots, T_n \in K}{\Gamma \vdash S \leq \text{Top}(K')\, T_1, \ldots, T_n} \tag{S-Top}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \leq S_1 \qquad \Gamma \vdash S_2 \leq T_2 \\ \Gamma \vdash S_1 \rightarrow S_2 \in \star\end{array}}{\Gamma \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \tag{S-Arrow}$$

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

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

$$\frac{\begin{array}{c}\{l_1, \ldots, l_n\} \subseteq \{k_1, \ldots, k_m\} \\ \text{for each } k_i = l_j, \;\; \Gamma \vdash S_i \leq T_j \\ \Gamma \vdash \{\!|k_1{:}S_1, \ldots, k_m{:}S_m|\!\} \in \star\end{array}}{\Gamma \vdash \{\!|k_1{:}S_1, \ldots, k_m{:}S_m|\!\} \leq \{\!|l_1{:}T_1, \ldots, l_n{:}T_n|\!\}} \tag{S-Record}$$

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

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

B.5 Typing

$$\frac{\Gamma \vdash e \in S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T} \tag{T-Subsumption}$$

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

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

$$\frac{\Gamma \vdash f \in T_1 \rightarrow T_2 \qquad \Gamma \vdash a \in T_1}{\Gamma \vdash f\, a \in T_2} \qquad \text{(T-ARROW-E)}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash T \sim \mathrm{Some}\,(A{\leq}U_1)\, U_2 \\ \Gamma \vdash S \leq U_1 \qquad \Gamma \vdash e \in [S/A]U_2\end{array}}{\Gamma \vdash \langle S, e\rangle{:}T \in T} \qquad \text{(T-SOME-I)}$$

$$\frac{\Gamma \vdash e_1 \in \mathrm{Some}\,(A{\leq}S_1)\, S_2 \qquad \Gamma, A{\leq}S_1, x{:}S_2 \vdash e_2 \in T}{\Gamma \vdash \mathrm{open}\; e_1 \;\mathrm{as}\; \langle A, x\rangle \;\mathrm{in}\; e_2 \;\mathrm{end} \in T} \qquad \text{(T-SOME-E)}$$

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

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

References

Abadi, M. 1993 (February). *Baby Modula-3 and a Theory of Objects.* Research Report 95. Digital Equipment Corporation, Systems Research Center, Palo Alto, California.

Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., & Moon, D. A. 1988. Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Notices*, **23**.

Bruce, K., & Mitchell, J. 1992 (January). 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. 1991 (March). The Equivalence of Two Semantic Definitions for Inheritance in Object-Oriented Languages. *In: Proceedings of Mathematical Foundations of Programming Semantics.*

Bruce, K. B. 1992 (January). *A Paradigmatic Object-Oriented Language: Design, Static Typing and Semantics.* Tech. rept. CS-92-01. Williams College.

Bruce, K. B. 1993 (January). Safe Type Checking in a Statically Typed Object-Oriented Programming Language. *In: Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages.*

Bruce, K. B., & Longo, G. 1990. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation*, **87**, 196–240. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (The MIT Press; to appear, 1993). An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

Bruce, K. B., & van Gent, R. 1993. *TOIL: A new Type-safe Object-oriented Imperative Language.* Submitted for publication.

Budd, T. 1991. *An Introduction to Object-Oriented Programming.* Reading, MA: Addison-Wesley.

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

Cardelli, L. 1986. Amber. *Pages 21–47 of:* Cousineau, G., Curien, P.-L., & Robinet, B. (eds), *Combinators and Functional Programming Languages.* Springer-Verlag. Lecture Notes in Computer Science No. 242.

Cardelli, L. 1988a. A Semantics of Multiple Inheritance. *Information and Computation,* **76**, 138–164. Preliminary version in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

Cardelli, L. 1988b (January). 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 (October). *Notes about* $F^{\omega}_{<:}$. Unpublished notes.

Cardelli, L. 1992a (January). *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* (The MIT Press; to appear, 1993).

Cardelli, L. 1992b (January). *Typed Foundations of Object-oriented Programming.* Tutorial given at POPL '92.

Cardelli, L., & Mitchell, J. 1991. Operations on Records. *Mathematical Structures in Computer Science*, **1**, 3–48. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (The MIT Press; to appear, 1993); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

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. 1991. An extension of system F with subtyping. *Pages 750–770 of:* Ito, T., & Meyer, A. R. (eds), *Theoretical Aspects of Computer Software (Sendai, Japan).* Lecture Notes in Computer Science, no. 526. Springer-Verlag.

Castagna, G., Ghelli, G., & Longo, G. 1992. A calculus for overloaded functions with subtyping. *Pages 182–192 of: ACM conference on LISP and Functional Programming.* San Francisco: ACM Press. Also available as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

Castagna, G. 1992 (May). *Strong Typing in Object-Oriented Paradigms.* Rapport de Recherche LIENS-92-11. Ecole Normale Supérieure, Paris.

Compagnoni, A. B., & Pierce, B. C. 1993 (August). *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. Submitted for conference publication.

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

Cook, W. R., Hill, W. L., & Canning, P. S. 1990 (January). 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* (The MIT Press; to appear, 1993).

Coppo, M., Dezani-Ciancaglini, M., & Venneri, B. 1981. Functional Characters of Solvable Terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, **27**, 45–58.

Curien, P.-L., & Ghelli, G. 1992. Coherence of Subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, **2**, 55–91. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (The MIT Press; to appear, 1993).

de Bruijn, N. G. 1972. Lambda-Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem. *Indag. Math.*, **34**(5), 381–392.

Ghelli, G. 1991 (October). A Static Type System for Message Passing. *Pages 129–143 of: Conference on Object-Oriented Programming Systems, Languages, and Applications.* Distributed as SIGPLAN Notices, Volume 26, Number 11, November 1991.

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.

Graver, J. O., & Johnson, R. E. 1990 (January). A Type System for Smalltalk. *Pages 125–135 of: Seventeenth Annual ACM Symposium on Principles of Programming Languages.*

Hofmann, M., & Pierce, B. 1994. A Unifying Type-Theoretic Framework for Objects. *In: Symposium on Theoretical Aspects of Computer Science.* Extended version available as "An Abstract View of Objects and Subtyping (Preliminary Report)," University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

Jategaonkar, L. A., & Mitchell, J. C. 1988 (July). ML with Extended Pattern Matching and Subtypes (preliminary version). *Pages 198–211 of: Proceedings of the ACM Conference on Lisp and Functional Programming.*

Kamin, S. 1988 (January). Inheritance in Smalltalk-80: A denotational definition. *Pages 80–87 of: Proceedings of the ACM Symposium on Principles of Programming Languages.*

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

Mitchell, J., Meldal, S., & Madhav, N. 1991 (January). An extension of Standard ML modules with subtyping and inheritance. *Pages 270–278 of: Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages.*

Mitchell, J. C. 1990 (January). 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* (The MIT Press; to appear, 1993).

Mitchell, J. C., Honsell, F., & Fisher, K. 1993 (June). A lambda calculus of objects and method specialization. *In: 1993 IEEE Symposium on Logic in Computer Science.*

Pierce, B. C., & Turner, D. N. 1993a (January). Object-Oriented Programming Without Recursive Types. *In: Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages (POPL).*

Pierce, B. C., & Turner, D. N. 1993b (April). *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.

Reddy, U. S. 1988 (July). Objects as Closures: Abstract Semantics of Object Oriented Languages. *Pages 289–297 of: Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming.*

Rémy, D. 1989. Typechecking records and variants in a natural extension of ML. *Pages 242–249 of: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*. ACM. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (The MIT Press; to appear, 1993).

Reynolds, J. C. 1983. Types, Abstraction, and Parametric Polymorphism. *Pages 513–523 of:* Mason, R. E. A. (ed), *Information Processing 83*. Amsterdam: Elsevier Science Publishers B. V. (North-Holland).

Reynolds, J. 1985. Three Approaches to Type Structure. *In: Mathematical Foundations of Software Development*. Springer-Verlag. Lecture Notes in Computer Science No. 185.

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* (The MIT Press; to appear, 1993).

Robinson, E., & Tennent, R. 1988 (October). *Bounded quantification and record-update problems*. Message to `Types` electronic mail list.

Snyder, A. 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. *Proceedings OOPSLA '86, distributed as ACM SIGPLAN Notices*, **21**(11), 38–45.

Ungar, D., & Smith, R. B. 1987. Self: The power of simplicity. *Pages 227–241 of: Proceedings of the ACM Symposium on Object-Oriented Programming: Languages, Systems, and Applications (OOPSLA)*.

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

Wand, M. 1988. Corrigendum: Complete Type Inference for Simple Objects. *In: Proceedings of the IEEE Symposium on Logic in Computer Science*.

Wand, M. 1989 (June). Type Inference for Record Concatenation and Multiple Inheritance. *Pages 92–97 of: Fourth Annual IEEE Symposium on Logic in Computer Science*.