Object-Oriented Programming Without Recursive Types

Benjamin C. Pierce* David N. Turner†

LFCS, University of Edinburgh

Abstract

It is widely agreed that recursive types are inherent in the static typing of the essential mechanisms of object-oriented programming: encapsulation, message passing, subtyping, and inheritance. We demonstrate here that modeling object encapsulation in terms of existential types yields a substantially more straightforward explanation of these features in a simpler calculus without recursive types.

1 Introduction

Static type systems for object-oriented programming languages have progressed significantly in the past decade. The line of research begun by Cardelli [11] and continued by Cardelli [18, 17, 14, 13], Mitchell [32, 10, 33], Bruce [8, 5, 7], and others [31, 39, 21, 10, 23, 20, 19, 26, 29, 44, 45, 46] has culminated in type-theoretic accounts [6, 14] of many of the features of languages like Smalltalk [28]. Our goal here is to reformulate the essential mechanisms of these accounts using a simpler type theory: we give a complete model of encapsulation, message passing, subtyping, and inheritance with `self` and `super`, using neither recursive types — which until now have generally been regarded as essential [6, 32] — nor the complex record extension operations that appear in some accounts.

The key step is an alternative treatment of encapsulation. Reynolds [41, 22] 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:

$$Point = Fix(P)\ \{getX : Int,\ setX : Int{\to}P\}$$

We choose type abstraction instead, following Mitchell and Plotkin's treatment of conventional abstract data types as existential types [34]:

$$\begin{aligned} Point = \exists(Rep)\ & \\ & \{state : Rep, \\ & \ methods : \{getX : Rep{\to}Int, \\ & \qquad\qquad\quad setX : Rep{\to}Int{\to}Rep\}\} \end{aligned}$$

Both the state of an object (the component *state*) and the fact that the methods operate on elements of its type are visible in this encoding, with the existential type protecting the state from external access except through the methods.

Section 2 develops our encoding of objects in more detail. In Section 3 we introduce subtyping, which captures the intuition that a class of objects may provide more services than are required by a particular context. Proper treatment of the interaction between encapsulation and subtyping requires a mechanism for expressing more refined subtyping constraints, which we obtain by extending subtyping to type operators. In Section 4 we show how inheritance can be provided within this framework. We then sketch a simple high-level syntax for object types and class declarations with a uniform translation into our basic calculus (Section 5) and close by comparing our work to other proposed static type systems for object-oriented programming in Section 6.

We develop our account using $F^{\omega}_{\leq}$, a polymorphic lambda-calculus with existential types and higher-order

*Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, U.K. Electronic mail: bcp@dcs.ed.ac.uk. Currently visiting INRIA, Projet Formel (Roquencourt).

†Same postal address; electronic mail: dnt@dcs.ed.ac.uk

bounded quantification, summarized in Appendix A. The examples in the paper were typeset using a prototype compiler for $F^{\omega}_{\leq}$ that typechecks and evaluates declarations preceded by the symbol >; they are followed, where appropriate, by the compiler's responses.

Basic familiarity with polymorphic type systems, subtypes, existential types, and conventional object-oriented languages is assumed; background reading on these topics can be found in [34, 18, 37, 11, 9, 28, 42]. A more detailed discussion of our constructions and some extensions appear in [37].

2 Objects

As in most type-theoretic accounts of object-oriented programming, we restrict our attention to the case of purely functional objects, in which methods must return a new copy of the internal state instead of updating it in place. For example, if `Rep` is the type of the internal state, then the type of the methods of a point object is:

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

The keyword `Fun` here abstracts the type variable `Rep`, yielding a function `PointM` from types to types that specifies the visible behavior of the point methods in terms of the abstracted representation type. An object satisfying this specification consists of a method vector of type `PointM(Rep)` for some concrete state type `Rep`, paired with a "current state" of type `Rep` both 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)};
```

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

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

(where "`*->*`" is the kind of functions from types to types). The type of point objects can now be reconstructed by applying the `Object` constructor to the specification `PointM`:

```
> Point == Object(PointM);
```

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 required, however, to give a proper treatment of the interaction between encapsulation and subtyping.

New point objects are created using the existential introduction form `pack`:

```
> p1 = pack
>        {state = {x=5},
>         methods = {setX = fun(s:{x:Int})
>                             fun(i:Int)
>                               {x=i},
>                    getX = fun(s:{x:Int})
>                             s.x}}
>      as Point
>      hiding {x:Int}
>      end;
p1 = <val> : Point
```

Note that, unlike many conventional object-oriented languages, the elements of an object type may have different internal representations and different implementations of their methods. This amounts to taking a structural view of object type equivalence instead of using "by-name" equivalence.

The `setX` method of a point `p` can be invoked as follows:

```
> Point'setX =
>   fun(p:Point)
>     open p as [Rep,r] in
>       fun(i:Int)
>         pack {state = r.methods.setX r.state i,
>               methods = r.methods}
>         as Point hiding Rep
>         end
>     end;
Point'setX = <val> : Point -> Int -> 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`; this ensures that the only functions applicable to the state component `r.state` are those in `r.methods`. We apply the `setX` function from `r.methods` to `r.state` and the new coordinate `i`, producing a new value of type `Rep`, which is re-packed as a point object with the same methods and hidden representation type as the original.

The conventional use of existential types in modeling abstract data types differs from their use here in modeling objects in that a "package" (an instance of an existential type) implementing an abstract data type is normally opened just once, as soon as it is created, whereas a package representing an object is kept closed except when it must be opened to send a message, and is then immediately re-packed as necessary.

In the recursive-records encoding of objects, sending a message to an object is usually modeled simply 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 that may be needed is placed on the receiver of a message, whereas we place it on the sender. (Fortunately, our method invocation functions can be generated automatically from the types of the methods; see Section 5.)

Since the `getX` method does not return a new state, no re-packing is necessary:

```
> 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` and `getX` to point objects:

```
> Point'getX (Point'setX p1 3);
3 : Int
```

3 Subtyping

Most object-oriented languages organize the specifications of objects into a subtype hierarchy, capturing the intuition that a class of objects may provide more services than are required by a particular context. 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,
>             getC: Rep->Color,
>             setC: Rep->Color->Rep};
> CPoint == Object(CPointM);
```

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 lambda-calculus with a subtype relation, writing $\Gamma \vdash S \leq T$ to mean that S is a subtype of T under assumptions Γ.

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

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

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

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

$\leq$

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

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

The subtype relation we use is a straightforward higher-order extension (due to Cardelli and Mitchell [13, 32]) of the subtype relation used in the calculus of second-order bounded quantification, $F_{\leq}$ [18, 16, 25]. Subtyping between type operators is defined pointwise: $\mathrm{Fun}\,(A{:}K)\,S$ is a subtype of $\mathrm{Fun}\,(A{:}K)\,T$ if $[U/A]S \leq [U/A]T$ for every type $U \in K$ — or, equivalently, if $S \leq T$ under no assumptions about the free variable A. This effectively extends the concept of subtyping to the interface specifications of objects, so that, for example, `CPointM` $\leq$ `PointM`.

The encoding of message passing in Section 2 must be generalized to interact properly with subtyping. For example, the typing of `Point'setX` allows it to be applied to an element of `CPoint` (using rules T-Subsumption and T-Arrow-E), but then the result of the application is an element of `Point`, rather than an element of `CPoint` as we would expect, since `Point'setX` is supposed to be a functional analog of the operation of sending the `setX` 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.

Moreover, conventional (second-order) bounded quantification, though it was originally introduced to handle just this sort of "update problem," does not provide a correct typing either. We might first attempt to generalize `Point'setX` as

```
wrong'setX =
  fun(P<Point)
    fun(p:P)
      open p as [Rep,r] in
         fun(i:Int)
            pack {state = r.methods.setX
                                r.state i,
                   methods = r.methods}
            as P hiding Rep
            end
      end;
```

(where `fun(P<Point)` denotes abstraction on a bounded type variable, sometimes written $\Lambda P{\leq}Point$). But this function is not well typed, as the reader can easily check. Indeed, a simple semantic argument [43] shows that the

only inhabitants of the type `All(P<Point) P->Int->P` are identity functions!

The crux of the difficulty is that `Point'setX` performs a functional update of the `state` field of `p`, while the quantifier `All(P<Point)` ranges over all subtypes of `Point`; in particular, the type of the `state` field is allowed to vary. It is well known that the types of updateable components of data structures must *not* be allowed to vary under the subtype relation. Fortunately, a natural way to obtain subtyping of the methods of an object type — the interface specification part — while fixing the type of the updateable state component is provided by the generalization of subtyping to type operators. An appropriate argument to `Point'setX` is an object whose method signature refines the signature of points — i.e., a value `v` such that `v : Object(M)`, where `M`$\leq$`PointM`:

```
> Point'setX =
>   fun(M<PointM)
>      fun(p:Object(M))
>         open p as [Rep,r] in
>            fun(i:Int)
>               pack {state = r.methods.setX
>                                r.state i,
>                     methods = r.methods}
>               as Object(M) hiding Rep
>               end
>         end;
Point'setX = <val>
           : All(M<PointM)
               (Object M) -> Int -> (Object M)
```

Applying this version of `Point'setX` to the type operator `CPointM` and an element of `CPoint` yields a `CPoint`, as desired.

Operator subtyping appears in most type theoretic accounts of object-oriented programming. Mitchell [32] uses it explicitly, while Cook, Hill, and Canning [23] and Bruce [6] rely on the closely related formalism of F-bounded quantification to achieve a similar effect.

4 Inheritance

The word "inheritance" is used to describe a variety of language features that allow object definitions to be constructed incrementally, sharing implementations of methods in hierarchies of related *classes*. In this section we demonstrate how inheritance can be implemented within the formal framework we have already developed for the more fundamental mechanisms of encapsulation and subtyping. For simplicity of exposition, we use a formulation of inheritance that makes the instance variables of a class visible to all of its subclasses, as in Smalltalk. A more refined style where instance variables are hidden from subclasses can also be supported in this framework; details appear in [37].

The `setX` and `getX` methods of colored points behave just like those of points. Inheritance allows these methods to be written once, in the definition of points, and then reused in the definition of colored points. To make the example more interesting, let us extend the interface of both points and colored points with a `slide` method that adds a given displacement to the current position:

```
> PointM == Fun(Rep) {getX: Rep->Int,
>                     setX: Rep->Int->Rep,
>                     slide: Rep->Int->Rep};
> CPointM == Fun(Rep) {getX: Rep->Int,
>                      setX: Rep->Int->Rep,
>                      slide: Rep->Int->Rep,
>                      getC: Rep->Color,
>                      setC: Rep->Color->Rep};
```

If `slide` is implemented in terms of the `setX` and `getX` methods, and if the colored point class provides a new implementation of `setX`, then this implementation should also be used by the inherited `slide` method. This *late binding* of recursive references in `slide` to `setX` and `getX` is what distinguishes the class of sliding points from a sliding point object. The class leaves recursive self-references unresolved, so that it can be extended to yield a subclass (which itself can be further extended); when the class is instantiated to form an object, the self-references are fixed and the methods all become concrete functions.

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

Note that classes themselves are values, not types. Moreover, the type of the point class is not a subtype of the type of the colored point class, although the type of colored point objects is still 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, `Rep`, so that we only need to deal with inheritance of the methods:

```
> Rep == {x:Int,color:Color};
```

The definition of a point class is then just a record of methods, abstracted on a record `self` of methods with the same types:

```
> pointClass =
>   fun(self:PointM(Rep))
>     {getX = fun(s:Rep) s.x,
```

```
>      setX = fun(s:Rep)
>               fun(i:Int) {x=i,color=s.color},
>      slide = fun(s:Rep) fun(d:Int)
>                self.setX s (plus (self.getX s) d)}
>     :<PointM(Rep)>;
pointClass = <val> : (PointM Rep) -> (PointM Rep)
```

The recursive references to `setX` and `getX` in `slide` 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 using `pack` as before:

```
> p = pack
>       {state = {x=1,color=red},
>        methods = rec [PointM(Rep)] pointClass}
>     as Object(PointM)
>     hiding Rep
>     end;
p = <val> : Object PointM
```

The class `pointClass` here has no superclasses: its behavior is defined directly. Colored points, on the other hand, should be 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 =
>   % superclass methods:
>   fun(super:PointM(Rep))
>   % recursively-defined "self" methods:
>   fun(self:CPointM(Rep))
>     {getX = super.getX,
>      setX = super.setX,
>      getC = fun(s:Rep) s.color,
>      setC = fun(s:Rep) fun(c:Color)
>               {x=s.x, color=c},
>      slide = super.slide}
>     :<CPointM(Rep)>;
buildCPointClass = <val>
                 :    (PointM Rep)
                   -> (CPointM Rep)
                   -> (CPointM Rep)
```

The `getX`, `setX`, and `slide` methods are inherited by copying them from the abstracted record of point methods `super`. 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 in this example.)

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 (which can be regarded as a record of point methods since `CPointM(Rep)` is a subtype of `PointM(Rep)`) obtained by taking a fixed point as before:

```
> cp =
>  pack
>   {state = {x=1,color=red},
>    methods = rec [CPointM(Rep)]
>                fun(self: CPointM(Rep))
>                  buildCPointClass
>                    (pointClass self) self}
>  as Object(CPointM)
>  hiding Rep
>  end;
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. For example, points should be represented using just an `x` field, while the representation of colored points also has a `color` field.

```
> PointR == {x:Int};
> CPointR == {x:Int, color:Color};
```

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

This 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 new record with a new `x` field and a copy of the old `color` field.

To resolve this difficulty, we need only observe 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`:

```
> 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},
>         slide = fun(s:FinalR) fun(d:Int)
>                   put s {x=(plus (get s).x d)}}
>        :<PointM(FinalR)>;
pointClass = <val>
```

```
          : All(FinalR)
                (FinalR->PointR)
             -> (FinalR->PointR->FinalR)
             -> (PointM FinalR)
             -> (PointM FinalR)
```

The get and put functions can be thought of as the functional analog of a pointer to the "pointClass part" of the state vector. This level of detail is typically hidden in type-theoretic accounts of object-oriented programming by extending the base calculus with operations for incrementally extending and modifying record valuess [44, 17, 14, 15].

Abstracting the method interface PointM and the "self representation type" PointR from the type of pointClass yields an operator describing the type of arbitrary class definitions.

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

The generic instantiation function new takes a class and an initial state and constructs an object using the fixed-point constructor as usual, fixing the "final representation" to be the same as the "self 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 < Top(*->*))
>   fun(SelfR < Top(*))
>   fun(selfClass: Class SelfM SelfR)
>   fun(s: SelfR)
>       pack {
>         state = s,
>         methods =
>           rec [SelfM(SelfR)]
>             (fun(self: SelfM(SelfR))
>               selfClass [SelfR]
>                 (fun(s:SelfR) s)
>                 (fun(s:SelfR) fun(s':SelfR) s')
>                 self)
>       } as Object(SelfM) hiding SelfR end;
new = <val>
    : All(SelfM<Top(*->*))
      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.

```
> extend =
>   % superclass interface:
>   fun(SuperM < Top(*->*))
>   % superclass representation:
>   fun(SuperR < Top(*))
>   % new class interface:
>   fun(NewM < SuperM)
>   % new class representation:
>   fun(NewR < Top(*))
>   % the superclass        :
>   fun(superClass: Class SuperM SuperR)
>   % "increment" function::
>   fun(inc:
>         % final representation:
>         All(FinalR)
>         % extractor:
>         (FinalR->NewR) ->
>         % overwriter:
>         (FinalR->NewR->FinalR) ->
>         % superclass methods:
>         (SuperM FinalR) ->
>         % self methods:
>         (NewM FinalR) ->
>         % ...returning the new methods:
>         (NewM FinalR))
>   % new->super extractor:
>   fun(get: NewR->SuperR)
>   % new<-super overwriter:
>   fun(put: NewR->SuperR->NewR)
>
>    % 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<Top(*->*))
         All(SuperR)
         All(NewM<SuperM)
         All(NewR)
              (Class SuperM SuperR)
           -> (All(FinalR)
                   (FinalR->NewR)
                 ->(FinalR->NewR->FinalR)
                 ->(SuperM FinalR)
                 ->(NewM FinalR)
                 ->(NewM FinalR))
           -> (NewR->SuperR)
           -> (NewR->SuperR->NewR)
           -> (Class NewM NewR)
```

Note that the type of `inc` is like a class type, except that it is also abstracted on the implementation of its superclass methods.

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},
>         slide = super.slide
>        })
>     (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'slide [CPointM] cp 3);
4 : Int
```

5 High-level Syntax

We have chosen to present our encodings in pure $F^{\omega}_{\leq}$, in order to clarify the precise type-theoretic implications of the basic mechanisms of object-oriented programming. Of course, the resulting encodings are too verbose to be of direct use in high-level object-oriented programming languages; some syntactic sugar is needed to make them palatable.

Our implementation of $F^{\omega}_{\leq}$ provides a simple high-level syntax for both object types and class declarations. The syntax of object types relies on the observation, developed in detail in [30], that message-sending functions like `Point'setX` can be generated uniformly, given the types of the methods: from the object declarations below, we automatically generate the types `PointM` and `Point` and the implementations of `Point'getX`, `Point'setX`, and `Point'slide`.

```
> Point ==
>   ObjectType(Rep) with
>     getX: Int,
>     setX: Int->Rep,
>     slide: Int->Rep
>   end;
PointM == Fun(Rep)
            {getX: Rep->Int,
             setX: Rep->Int->Rep,
             slide: Rep->Int->Rep}
Point == Object PointM
Point'getX : All(M<PointM) (Object M) -> Int
Point'setX : All(M<PointM)
               (Object M) -> Int -> (Object M)
Point'slide : All(M<PointM)
                (Object M) -> Int -> (Object M)

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

Note that each of the definitions of `Point` and `CPoint` generates functions for sending the `setX`, `getX`, and `slide` messages. In this example, those for `CPoint` are redundant, since they have exactly the same behavior and essentially the same typing. In general, however, `CPoint`'s version of a given method may have a more specific type than `Point`'s; in this case, both message-sending functions may be useful.

High-level class definitions are used to generate the arguments to the `extend` function and to supply convenient notation for accessing and updating instance variables. For example, the following high-level definitions of the point and colored point classes have exactly the same behavior as the low-level implementations developed in the previous section:

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

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

We require the inherited methods `setX`, `getX`, and `slide` to be named explicitly here in order to make the translation from high-level to low-level syntax completely local. A more convenient syntax can be obtained by maintaining information about methods and instance variables between class and object type declarations. This extension can be viewed as the introduction, in the high-level syntax, of a simple form of extensible records [44, 17, 40] .

6 Related Work

The recent accounts of object-oriented programming that are closest to ours in spirit are those due to Bruce [6] and Cardelli [14].

Bruce [6] develops a formal semantics (based on previous models by Mitchell [32], Cook [23], 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 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..." [6, abstract]

While we agree that fixed points at the element level are required to model the inheritance of methods referring to *self*, we contend that the complexity of constructions based on 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 [40, 17] 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 [44, 14] is not needed.

Cardelli's treatment of object-oriented programming [14, 15] aims to describe the same basic features of encapsulation, subtyping, and inheritance as ours and Bruce's. Like us (and unlike Bruce), Cardelli adopts a mainly syntactic point of view, trying to capture a set of fundamental requirements in the form of a typed lambda-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, indeed, can be encoded using only universal quantification).

Perhaps the most significant difference between our approach and those of Bruce and Cardelli lies in the treatment of *multi-methods* — operations like the union of sets, whose implementation must obtain simultaneous access to the internal representations of two or more objects. Since our method signatures are always essentially unary functions of the representation type, such operations cannot be represented as ordinary methods in our framework. Bruce and Cardelli do allow such methods in a restricted form, where the implementation of a multi-method may obtain concrete access to the representation of only one object, other arguments of the same type being accessible through their public interface. In [38], we show how a more general case can be handled by associating multi-methods with classes rather than objects, using Cardelli and Wegner's *partially abstract types* [18] to enforce proper encapsulation.

Recent papers by Castagna, Ghelli, and Longo [20, 19, 26] propose an intriguing new approach to the foundations of object-oriented programming. Taking overloading and subtyping as basic, instead of encapsulation and subtyping, they develop an underlying calculus that promises to model some features — notably the *generic functions* of languages such as CLOS [3] — that fall completely outside the scope of previous theories, including ours. However, their account does not yet include some of the characteristic features of object-oriented programming, encapsulation in particular, that have motivated our constructions.

7 Future Work

The extension of purely functional accounts of typing for object-oriented programs to allow the modification of instance variables in-place is a relatively straightforward matter. Given an extension of the base calculus with mutable reference cells, we simply choose (by convention) to let the state component of objects consist of records of reference cells and to make the result type of most methods be `Unit`. There is one problem, though: languages with references nearly always choose the call-by-value evaluation order rather than the call-by-name strategy we have assumed here. But the fixed point combinator used in the `new` function in Section 4 only makes sense in a language with call-by-name evaluation; the usual fixed point combinator at functional types that is found in call-by-value languages causes `new` to fail to terminate in some cases. This deficiency, which seems to be common to all lambda-calculus models of object-oriented programming (except when a compilation function is involved, as in [6, 32], so that the appropriate fixed points in the semantic model can be chosen without the need to give a type for a polymorphic fixed point operator), has also been pointed out by Wand [46] and Cook, Hill, and Canning [23]. To retain the syntactic flavor of our approach, it appears to be necessary to introduce either more sophisticated record types, so as to be able to write the type of a polymorphic fixed point operator applicable to an arbitrary record of functions, or else a lazy record constructor.

A useful variant of inheritance allows classes to specify some methods as "private"; these methods are visible in the implementations of inheriting subclasses but hidden otherwise; the implementation of inheritance given in Section 4 can be extended straightforwardly to include this feature.

A model of *delegation-based* object-oriented programming [4] related to the one proposed by Mitchell [32] can easily be formed by modifying our constructions so that the application of the fixed-point operator to the method vector of an object is performed at message-sending time rather than within the `new` function.

More speculatively, it may be possible to extend our approach to include multiple inheritance by working in a variant of $F^{\omega}_{\leq}$ with intersection types [24]. A second-order fragment of this calculus is studied in [35].

Acknowledgements

We are grateful for conversations with Dave Berry, Kim Bruce, Luca Cardelli, Giuseppe Castagna, William Cook, Giorgio Ghelli, Bob Harper, Robin Milner, John Mitchell, Kevin Mitchell, Didier Rémy, Nick Rothwell, and Phil Wadler, and especially to Martin Hofmann for important insights into the formal background of our constructions. Martin Hofmann and Kevin Mitchell suggested a number of improvements in an earlier draft.

References

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

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

[3] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification X3J13 document 88-002R. *SIGPLAN Notices*, 23, 1988.

[4] A. H. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 30–46, 1986.

[5] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991.

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

[7] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

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

[9] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

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

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

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

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

[14] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992.

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

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

[17] Luca Cardelli and John Mitchell. Operations on records (summary). In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of Fifth International Conference on Mathematical Foundations of Programming Language Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, Tulane University, New Orleans, March 1989. Springer Verlag. To appear in Mathematical Structures in Computer Science; also available as DEC Systems Research Center Research Report #48, August, 1989.

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

[19] Giuseppe Castagna. Strong typing in object-oriented paradigms. Rapport de Recherche LIENS-92-11, Ecole Normale Supérieure, Paris, May 1992.

[20] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions and subtyping. Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris, February 1992.

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

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

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

[24] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

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

[26] Giorgio Ghelli. A static type system for message passing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 129–143, Phoenix, Arizona, October 1991. Distributed as SIGPLAN Notices, Volume 26, Number 11, November 1991.

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

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

[29] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990.

[30] Martin Hofmann and Benjamin Pierce. An abstract view of objects and subtyping (preliminary report). Technical Report ECS-LFCS-92-226, University of Edinburgh, LFCS, 1992.

[31] S. Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 80–87, January 1988.

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

[33] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 270–278, Orlando, FL, January 1991.

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

[35] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

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

[37] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. Submitted to the Journal of Functional Programming, August 1992.

[38] Benjamin C. Pierce and David N. Turner. Statically typed multi-methods via partially abstract types. Draft technical teport, September 1992.

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

[40] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 242–249. ACM, January 1989.

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

[42] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development.* Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[43] Edmund Robinson and Robert Tennent. Bounded quantification and record-update problems. Message to `Types` electronic mail list, October 1988.

[44] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[45] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[46] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

A Summary of $F^{\omega}_{\leq}$

This appendix summarizes the syntax and typing rules of the typed λ-calculus $F^{\omega}_{\leq}$, an extension of Girard's system F^{ω} [27] to include subtyping. The ideas behind this system are due to Cardelli, particularly to his 1988 paper, "Structural Subtyping and the Notion of Power Type" [12]; the extension of the subtype relation to type operators was developed by Cardelli and Mitchell [13, 32, 8, 1]. Cardelli [13] has given a more powerful treatment of operator subtyping, including both monotonic and antimonotonic subtyping in addition to pointwise subtyping. The version of $F^{\omega}_{\leq}$ used here was obtained as an instance of the definition of a "Generalized Subtyping System" [36], a generalized type system in the sense of Barendregt [2] extended with Cardelli's powerkinds. A model for a variant of $F^{\omega}_{\leq}$ with recursive types has been given by Bruce and Mitchell [8], but its metatheory remains largely unstudied.

A.1 Syntax

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

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

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

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

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

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

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

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

A.2 Contexts

$$\vdash \bullet \ \text{context} \tag{C-Empty}$$

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

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

A.3 Kinding

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

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

$$\frac{\Gamma \vdash S \in K_1{\rightarrow}K_2 \qquad \Gamma \vdash T \in K_1}{\Gamma \vdash S\,T \in K_2} \tag{K-Arrow-E}$$

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

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

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

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

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

A.4 Subtyping

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash S_1 \leq T_1 \qquad \Gamma, A{\leq}S_1 \vdash S_2 \leq T_2 \qquad \Gamma \vdash \mathrm{Some}\,(A{\leq}S_1)\,S_2 \in \star}{\Gamma \vdash \mathrm{Some}\,(A{\leq}S_1)\,S_2 \leq \mathrm{Some}\,(A{\leq}T_1)\,T_2} \quad (\textsc{S-Some})$$

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

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

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

A.5 Typing

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e \in \{l{:}T\}}{\Gamma \vdash e.l \in T} \quad (\textsc{T-Record-E})$$