Statically Typed Friendly Functions via Partially Abstract Types

Benjamin C. Pierce
INRIA-Roquencourt
Benjamin.Pierce@inria.fr

David N. Turner
University of Edinburgh
David.Turner@dcs.ed.ac.uk

University of Edinburgh Technical Report
ECS-LFCS-93-256

and

INRIA-Roquencourt Rapport de Recherche
No. 1899

April 1993

Abstract

A well-known shortcoming of the object model of Simula and Smalltalk is the inability to deal cleanly with methods that require access to the internal state of more than one object at a time. Recent language designs have therefore extended the basic object model with notions such as *friends' methods* and *protected features*, which allow external access to the internal state of objects but limit the scope in which such access can be used. We show that a variant of this idea can be added to any type-theoretic model of the basic object-oriented mechanisms (encapsulation, message passing, and inheritance), using a construction based on Cardelli and Wegner's *partially abstract types*, a refinement of Mitchell and Plotkin's type-theoretic treatment of abstract types.

1 Introduction

The definition of the word "object" in programming languages descended from Simula and Smalltalk might be phrased as follows:

> "An object is a piece of state and a collection of methods for manipulating this state, protected by an abstraction barrier that prevents external access to the state except through the methods."

This straightforward object model leads to elegant programming languages and a natural style of programming with objects, but it has some serious shortcomings. Chief among these is the corollary that each method has access to the internal state of only one object. There are some important situations in which this restriction is unacceptable.

To take a standard example, suppose we want to define a class of objects representing finite sets of integers, supporting efficient operations for insertion of new elements, testing membership, and set union. If we implement the internal state of each set object as a balanced tree, then the insertion and membership methods can traverse and modify the tree in the usual way. However, to implement set union efficiently, we need to have access to the balanced tree representations of *both* of the sets involved. But this is not possible: the object model specifies that each method is associated with exactly one object and has access to the internal state of just that object.

Languages such as CLOS [BDG$^+$88] have addressed this problem by dropping the strong encapsulation requirement imposed by Smalltalk's object model. In CLOS, the internal state of an object is available *globally*. This facilitates the implementation of functions like set union, at the cost of weaker linguistic support for data abstraction. The restriction that objects should normally be manipulated by sending them messages, rather than by modifying their internal state directly (using their slot accessors), is a matter of programming style: it is not enforced by the language.

Other recent language designs take a more refined approach. In C++ [Str86], each class definition may include a group of *friend functions*, which live outside of the instances of the class — they are functions, not methods — but are allowed to access the internal state of objects of that class on the same basis as the methods. Eiffel's *selective exports* provide a more general capability: each class may name a collection of classes whose methods are allowed direct access to its instances' internal states. This idea is refined still further in Chambers' language Cecil [Cha92], in which multi-methods are regarded as "part of" all of the objects to which they may apply. Rouaix's Alcool language [Rou90b, Rou90a, Rou92] explores a related approach based on abstract types and overloading.

In this paper, we develop a statically typed object model supporting flexible access to internal object states and enforcing encapsulation. This work extends recent theoretical work on static type systems for Smalltalk-style objects [Mit90, CCH$^+$89, Bru93, Bru92, Car92, PT93, MHF93]. Indeed, the same ideas can be used to extend *any* of these models with friendly functions. In outline, the construction is as follows:

- We begin with Smalltalk's overly restrictive but extremely simple object model, in which access to the internal state is limited to an object's methods.
- Efficient access is supported within this object model by providing each object with a method that returns its whole state. For example, the interface of set objects includes a method `rep`, which directly returns the set's internal representation as a balanced tree. The union function may now be implemented straightforwardly by sending `rep` to both arguments and efficiently merging the balanced trees.
- Finally, the `rep` method is hidden by encapsulating both the class definition of sets and the implementation of the `union` function in an abstract type. The technical device used here to hide just the `rep` method while leaving the others visible outside the scope of the abstract type is a natural generalization of Cardelli and Wegner's notion of *partially abstract type* [CW85].

A bonus of our approach is that the extra mechanism affects only the creation of objects and the invocation of friendly functions on them. Programs that manipulate objects in terms of the original object model are unaffected. Even the relatively intricate mechanism of inheritance can be extended straightforwardly to support inheritance of friendly functions.

It is important to note that we are dealing here only with the problem of encapsulating operations that require access to the state of several objects *of the same class*, not with the more general problem of *multiple-dispatch*, where the run-time types of several parameters to a generic function call can be used to determine which method body is executed, and where access to the internal representations of several objects of different classes may be required in the implementation of methods. Moreover, we do not propose that friendly functions should necessarily be considered as *methods* and invoked by sending messages to objects; here, for clarity of exposition, they are presented simply as groups of functions associated with classes. We use the phrase *friendly functions* rather than *multi-methods* to avoid confusion with object models involving multiple-dispatch.

In Section 2 we summarize the relevant parts of a static type system for objects originally presented in [PT93]. Section 3 describes partially abstract types and their encoding in terms of bounded existential types; Section 4 describes their use in encoding friendly functions. The encoding of inheritance found in [PT93] can also be modified to work in the presence of friendly functions; the construction is described in Appendix C.

Our model of objects is based on $F^{\omega}_{\leq}$, 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 is assumed; background reading on these topics can be found in [MP88, CW85, PT93, Car88a, Bud91, GR83, Rey85]. Although our construction does not depend on the details of any particular type-theoretic model of objects, familiarity with the development in [PT93] will be helpful for understanding the more technical parts of the paper.

2 Objects

A number of type-theoretic treatments of objects and message passing are available in the recent literature [Mit90, CCH+89, Bru93, Bru92, Car92, PT93]. For the sake of concreteness, we develop our account of friendly functions using a particular model of objects that we have discussed in depth in [PT93]; however, the ideas here are not sensitive to the details of this model.

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 a one-field record:

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

A method is a function that implements a transformation on the state. To simplify the underlying formal model, we use a functional style of programming with objects where, instead of updating the state in-place, a method returns a new state. (Questions of typing are not affected by this simplification; the model can be extended to include imperative-style object-oriented programming [Bru93].) 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 : *->*
```

(The *kind* `*->*` printed by the typechecker expresses the fact that `PointM` maps types — elements of the kind `*` — to types.) 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 is formed by packing its state together with its methods using the constructor `new_object`. Given the type of the state, a type function describing the methods, a starting state, and a group of method implementations, `new_object` constructs a new object. For example, a point object based on the representation type `{|x:Int|}` can be created as follows:

```
# PointR = {|x: Int|};
PointR : *
# new_object
#   PointR    % State type
#   PointM    % Type function describing methods
#   {x=5}     % Starting state
#             % Record of method implementations:
#   {bump = fun(state: PointR) {x = plus 1 state.x},
#    setX = fun(state: PointR) fun(newX:Int) {x = newX},
#    getX = fun(state: PointR) state.x};
<val> : Object PointM
```

The type of the new point object, `Object PointM`, is formed by applying the object type constructor `Object` to the interface specification of the point methods. In general, `Object M` is the type of objects with interface M.

For each method of an object, we provide a function that implements the operation of sending a message to invoke that method. For example, point objects come with three message-sending functions:

```
send_bump : All(M<PointM) Object M -> Object M
send_setX : All(M<PointM) Object M -> Int -> Object M
send_getX : All(M<PointM) Object M -> Int
```

The types of these functions are simple generalizations of the following simpler typings:

```
send_bump : Object PointM -> Object PointM
send_setX : Object PointM -> Int -> Object PointM
send_getX : Object PointM -> Int
```

The extra quantification `All(M<PointM)` allows for the possibility of subtyping among object types: instead of specifying that `send_bump`, for example, maps points to points, we specify that it maps objects with interface `M` to objects with the same interface `M`, whenever `M` is more specialized than `PointM`. The relation "more specialized than" is captured here by the notion of *subtyping* (defined in more detail in Appendices A and B). For example, `CPointM` (below) is a subtype of `PointM` and so `send_bump` can also be applied to *colored* one-dimensional point objects, which are elements of `Object CPointM`.

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

To complete the example, we can construct a colored point object and send it the messages `bump` and `getX` using the point message-sending functions `send_bump` and `send_getX`:

```
# CPointR = {|x: Int, c: Color|};
CPointR : *
# c = new_object
#   CPointR     % State type
#   CPointM     % Type function describing methods
#   {x=5,c=red} % Starting state
#   % Record of method implementations:
#   {bump = fun(state: CPointR) {x = plus 1 state.x, c = state.c},
#    setX = fun(state: CPointR) fun(newX: Int) {x = newX, c = state.c},
#    getX = fun(state: CPointR) state.x,
#    setC = fun(state: CPointR) fun(newC: Color) {x = state.x, c = newC},
#    getC = fun(state: CPointR) state.c};
c = <val> : Object CPointM
# c' = send_bump CPointM c;
c' = <val> : Object CPointM
# send_getX CPointM c';
6 : Int
```

Complete details of the implementation of the `Object` type constructor, the `new_object` value constructor, and the functions for sending messages can be found in [PT93]. We turn now to a review of the key type-theoretic mechanisms on which our account of friendly functions rests: Mitchell and Plotkin's use of existential types to give a straightforward type-theoretic explanation of abstract types and Cardelli and Wegner's extended notion of *partially abstract types*.

3 Partially abstract types

Mitchell and Plotkin observed that abstract types can be modeled by existential types in polymorphic lambda-calculi [MP88]. For example, consider the following abstract type of counters, encoded as an existential type:

```
# CounterPackage =
#   Some(Rep) {|
#     initial: Rep,
#     inc: Rep -> Rep,
#     dec: Rep -> Rep,
#     isZero: Rep -> Bool
#   |};
CounterPackage : *
```

The type `Rep` is completely abstract: the only operations available on it are `initial`, `inc`, `dec` and `isZero`. We choose, for the purposes of illustration, an implementation of counters based on the following concrete representation type (the `zero` field records whether the counter is zero):

```
# CounterR = {|count: Int, zero: Bool|};
CounterR : *
```

One possible implementation of counters is:

```
# counterImpl =
#    {initial = {count = 0, zero = true},
#     inc = fun(c:CounterR)
#       let new = plus c.count 1
#       in {count = new, zero = eqInt new 0} : CounterR
#       end,
#     dec = fun(c:CounterR)
#       let new = minus c.count 1
#       in {count = new, zero = eqInt new 0} : CounterR
#       end,
#     isZero = fun(c:CounterR) c.zero};
counterImpl = <val>
            : {|initial: {|count:Int, zero:Bool|},
                inc: CounterR->CounterR,  dec: CounterR->CounterR,
                isZero: CounterR->Bool|}
```

(The two annotations `:CounterR` affect only the way that types are printed; if they were omitted, the typechecker would give the type of `inc` and `dec` in the equivalent but less readable form `CounterR->{|count:Int,zero:Bool|}`.) This concrete implementation can be encapsulated as an instance of the existential type `CounterPackage` by packing it with its "witness type" `CounterR`:

```
# counterPack = <CounterR, counterImpl> : CounterPackage;
counterPack = <val> : CounterPackage
```

Given an encapsulated implementation such as `counterPack`, its components can be accessed using an `open` expression. The abstract representation type is bound to the variable `Rep` and the record of operations is bound to `counter`, allowing us to use the counter operations within the scope of the `open`:

```
# open counterPack as <Rep,counter> in
#   counter.isZero(counter.inc counter.initial)
# end;
false : Bool
```

In a type theory with a notion of subtyping, existential types can be refined to declare a bound for the hidden representation type. Cardelli and Wegner [CW85] show how bounded existential types can be used to implement *partially abstract types*, where the exact representation type is hidden but some of its structure may be revealed. For example, the type `CounterPackage` can be modified to reveal the fact that the representation of a counter is a record containing a field `zero` of type `Bool`, without revealing the other components of the representation:

```
# NewCounterPackage =
#   Some(Rep < {|zero:Bool|}) {|
#     initial: Rep,
#     inc: Rep -> Rep,
#     dec: Rep -> Rep,
#     isZero: Rep -> Bool
#   |};
NewCounterPackage : *
```

An instance of the type `NewCounterPackage` can be created using the same implementation as before, by changing the interface type `CounterPackage` to `NewCounterPackage`:

```
# newCounterPack = <CounterR, counterImpl> : NewCounterPackage;
newCounterPack = <val> : NewCounterPackage
```

This refinement enables us to access a counter's `zero` field directly, instead of using the `isZero` function:

```
# open newCounterPack as <Rep,counter> in
#   (counter.inc counter.initial).zero
# end;
false : Bool
```

On the other hand, since we do not have access to all of the counter representation, `counter.initial` is still the only way to create a counter, and `counter.inc` and `counter.dec` are the only ways to increment and decrement a counter.

4 Friendly functions

Equality functions are a commonly used example of friendly functions. Suppose, for instance, that we wish to extend the point objects of Section 2 to allow an equality function to be defined. We can provide access to the internal state by adding a method, `rep`, which returns the internal state of an object:

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

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

The implementation of equality relies on a function `getRep`, which, given an internal state type `R` and an object `ob` containing a `rep` method (with result type `R`), extracts the internal state of `ob` by sending it the `rep` method.

```
# getRep;
<val> : All(R) (Object (Fun(Rep){|rep:Rep->R|})) -> R
```

(Since the implementation of `getRep` depends on details of the representation of objects, we show only its type.)

We can build an initial object `init` satisfying the interface type `PointI` as before. We package the initial object together with its equality function, `eq`, which uses the `getRep` function to access the internal state of both of its arguments:

```
# pointImpl =
#   {init =
#      new_object
#        PointR      % State type
#        PointI      % Type function describing methods
#        {x = 0}     % Starting state
#                    % Record of method implementations:
#        {bump = fun(s: PointR) {x = plus 1 s.x},
#         getX = fun(s: PointR) s.x,
#         setX = fun(s: PointR) fun(i: Int) {x = i},
#         rep =  fun(s: PointR) s},
#    fns = {
#      eq =
#        fun(p: Object PointI) fun(q: Object PointI)
#          eqInt (getRep PointR p).x (getRep PointR q).x
#      }
#   };
pointImpl = <val>
          : {|init: Object PointI,
              fns: {|eq:(Object PointI)->(Object PointI)->Bool|}|}
```

Note that `getRep PointR` can be applied to an object of type `Object PointI`, since `PointI` is a subtype of `Fun(Rep){|rep:Rep->PointR|}`.

The only problem with the above implementation is that we have lost some modularity, since there are no restrictions on the usage of the `rep` method. Since the interface `PointI` exposes `rep`, anyone can invoke the `rep` method on an element of `Object PointI`..

The crucial observation is that we can use a *higher-order bounded existential quantifier* to build an encapsulation barrier that prevents the `rep` method being used except in the implementation of equality:

```
# pointPackage =
#   <PointI, pointImpl>
#   : Some(PointI < PointM) {|
#       init: Object PointI,
#       fns: {|eq: Object PointI -> Object PointI -> Bool|}
#     |};
pointPackage = <val>
             : Some(PointI<PointM)
                 {|init: Object PointI,
                   fns: {|eq:(Object PointI)->(Object PointI)->Bool|}|}
```

Because of the bound `PointI<PointM`, elements of type `Object PointI` can be manipulated using the message-sending functions for points:

```
# open pointPackage as <PointI, point> in
#   let p = point.init in
#     point.fns.eq p (send_setX PointI p 1)
#   end
# end;
false : Bool
```

Note that, since the actual interface type `PointI` is hidden, `point.init` is the only way to create objects of type `Object PointI`.

For readers familiar with the literature on type-theoretic models for object-oriented languages, we should note that the equality function on one-dimensional points is not the clearest example of a binary function. Since the internal state of points — just a single integer — is already fully accessible through the operations `setX` and `getX`, this example does not distinguish between formalisms that allow full-fledged friendly functions, with privileged access to the states of more than one object, from formalisms such as [Mit90, CCH+89, Bru93, Car92], which allow only a weak form of binary methods that can accept additional arguments of the same object type as the receiver but can access only one of them concretely. With this in mind, however, we retain the example here for brevity, and for ease of comparison with previous work.

This completes the main body of the development. We have shown how to extend the basic object model of Smalltalk with friendly functions by a simple construction based on partially abstract types. Since bounded existential quantifiers can be encoded already in a second-order lambda-calculus with bounded quantification, it follows that this construction is available in any of the type-theoretic encodings of objects based on extensions of this calculus [Mit90, CCH+89, Bru93, Bru92, Car92, PT93, MHF93] with no complication of the underlying formal theory.

One question that immediately arises is whether this construction interacts smoothly with mechanisms of *inheritance*. Somewhat surprisingly, we find that it does: for example, an implementation of inheritance based on the object model of [PT93] can be extended fairly straightforwardly with friendly functions. Indeed, the friendly functions themselves can be inherited by subclasses! Since the details of this development are rather technical, we omit them here; interested readers may find them in Appendix C.

5 Future Work

We believe that the high-level syntax for class definitions developed in [PT93] can be extended to provide a convenient syntax for class definitions with friendly functions.

In a more theoretical vein, we are intrigued by the possibility of combining the techniques presented here with recent work on formalising CLOS's object model [Ghe91, CGL92, Cas92]. The required theoretical basis, an extension of Castagna, Ghelli, and Longo's lambda-calculus with subtyping and overloading to include $F^{\omega}_{\leq}$'s higher-order polymorphism, would be of significant interest in its own right.

Acknowledgements

We are grateful to Martín Abadi, Luca Cardelli, Ralph Johnson, Oscar Nierstrasz, and François Rouaix for productive conversations. Adolfo Scorro and Oscar Nierstrasz gave us useful pointers into the literature.

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. Benjamin Pierce was hosted by INRIA-Roquencourt during the fall of 1992 and spring of 1993, when most of this report was written.

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 [CW85] with a notion of type operator familiar from Girard's system F^ω [Gir72]. 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* [CW85] 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 type failure. 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. [Car86]) for subtyping between record types:

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

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:

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

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* [CW85]).

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* [Gir72], 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
```

This example declares the type `T` and uses 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 `*`, so we could have written `Fun(A) Fun(B) {| fst: A, snd: B |}` here.)

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 [Gir72] with subtyping. The ideas behind this system are due to Cardelli, particularly to his 1988 paper, "Structural Subtyping and the Notion of Power Type" [Car88b]; the extension of the subtype relation to type operators was developed by Cardelli and Mitchell [Car90, Mit90, BM92]. Cardelli [Car90] 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 [BM92].

B.1 Syntax

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

K	$::=$	$\star$	kind of types
	$\mid$	$K \rightarrow K$	kind of type operators
T	$::=$	A	type variable
	$\mid$	$\mathrm{Fun}\,(A{:}K)\,T$	type operator
	$\mid$	$T\ T$	application of a type 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
	$\mid$	$\{\lvert l_1{:}T_1, ..., l_n{:}T_n \rvert\}$	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
	$\mid$	$\{l_1 = e_1, ..., l_n = e_n\}$	record construction
	$\mid$	$e\,.\,l$	field selection
Γ	$::=$	$\bullet$	empty context
	$\mid$	$\Gamma,\ x{:}T$	variable binding
	$\mid$	$\Gamma,\ A{\leq}T$	type variable binding with bound

B.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

B.1.3. Convention: Whenever we write Γ, $A{\leq}T$ or Γ, $x{:}T$ 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 [dB72], 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.

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

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

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

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

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

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

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

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

B.4 Subtyping

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

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

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

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

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

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

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

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

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

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

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

B.5 Typing

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

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

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

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

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

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

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

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

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

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

C Inheritance

In this appendix we show that the implementation of inheritance developed in [PT93] can be extended to deal with friendly functions. Although our implementation provides inheritance of friendly functions themselves, this is only to illustrate the flexibility of our approach, rather than to suggest that a language design based on this form of inheritance would be ideal in practice. In fact, none of the details of this implementation of inheritance should be regarded as crucial; we present them in full only for the sake 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. Other choices are possible; see [PT93].

We reuse our example of points and colored points to illustrate our implementation of inheritance; for brevity, we omit the `bump` method in this section. Consider the interface for colored points:

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

Suppose we wish the `setX` and `getX` methods of colored points to 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. We adapt our previous encoding of inheritance in [PT93] to automatically package up any friendly functions relevant to an object.

A *class* is a data structure that can either be used as the starting point for the definition of subclasses by incremental extension (using the `extend` function), or can be instantiated to a package using the `instantiate` function. The package may then be opened and used as in Section 4. Graphically:

$$\begin{tikzcd}
\texttt{pointClass} \arrow[r, "\texttt{instantiate}"] \arrow[d, "\texttt{extend}"] & \texttt{pointPackage} \arrow[r, "\texttt{open}"] & \texttt{<PointM,point>} \\
\texttt{cpointClass} \arrow[r, "\texttt{instantiate}"] & \texttt{cpointPackage} \arrow[r, "\texttt{open}"] & \texttt{<CPointM,cpoint>}
\end{tikzcd}$$

Note that the type of `cpointClass` will not be a subtype of the type of `pointClass`, although the type of colored point objects is a subtype of the type of point objects, as expected.

For notational convenience we begin by generalizing the type of `pointPackage` in Section 4 to yield a general `Package` type constructor:

```
# Package =
#   Fun(SelfM: *->*)  % Method interface
#   Fun(SelfF: *->*)  % Friendly function interface
```

```
#     Some(SelfI<SelfM)              % Hidden internal interface
#      {|init: Object SelfI,         % An initial "new" object
#        fns: SelfF(Object SelfI)|}; % Class friendly functions
Package : (*->*)->(*->*)->*
```

Note that the friendly function interface, `SelfF`, is a type operator which is abstracted on the type of the whole object, `Object SelfI`, not the representation type (as is the case for method interface types such as `SelfI`). For example, the friendly function inferface for point objects is:

```
# PointF = Fun(Obj) {|eq: Obj->Obj->Bool|};
PointF : *->*
```

The result of instantiating a class with all the information it needs to form a package is a record containing two fields: `methods`, a record of method implementations (including the `rep` method, by convention) and `fns`, a record of functions that implement the friendly functions of the class:

```
# ClassResult =
#   Fun(SelfI: *->*) % Method interface including rep method
#   Fun(SelfF: *->*) % Friendly function interface
#   Fun(FinalR)      % Final representation
#     {| methods: SelfI FinalR, fns: SelfF(Object SelfI) |};
ClassResult : (*->*)->(*->*)->*->*
```

The type of a class is more complicated. To save space, we refer the reader to [PT93] for an explanation.

```
# Class =
#   Fun(SelfM: *->*) % object interface type
#   Fun(SelfI: *->*) % object interface extended with rep method
#   Fun(SelfF: *->*) % friendly function interface
#   Fun(SelfR)       % internal representation type
#     All(FinalR)                     % final representation
#     (FinalR->SelfR) ->              % extractor
#     (FinalR->SelfR->FinalR) ->      % overwriter
#     (SelfM FinalR) ->               % self methods
#     ClassResult SelfI SelfF FinalR; % Class result
Class : (*->*)->(*->*)->(*->*)->*->*
```

A class leaves recursive self-references unresolved, so that it can be extended to yield a subclass (which itself can be further extended); when a class is instantiated to form a package, the self-references are fixed and the methods all become concrete functions. The instantiation function `instantiate` takes an initial state and a class and constructs the actual methods `self.methods` and functions `self.fns` using the fixed-point constructor `rec`. It fixes the "final representation" to be the same as the "self representation" and supplies an identity function as the extractor and, as the overwriter, a two-argument function that simply returns its second argument. The methods `self.methods` are then packaged as an object in the usual way. The complete result is then encapsulated as a `Package`, hiding the internal interface type `SelfI`.

```
# instantiate =
#   fun(SelfM:*->*) % object interface type
#   fun(SelfI < SelfM)     % object interface extended with rep method
#   fun(SelfF:*->*) % friendly functions
#   fun(SelfR:*)    % internal representation type
#   fun(s: SelfR)          % initial state
#   fun(selfClass: Class SelfM SelfI SelfF SelfR)
#     let self =
#       rec (ClassResult SelfI SelfF SelfR)
#         fun(self: ClassResult SelfI SelfF SelfR)
#           selfClass SelfR (fun(s: SelfR) s)
#           (fun(s: SelfR) fun(s': SelfR) s') self.methods
#     in
#       <SelfI,
#        {init = new_object SelfR SelfI s self.methods,
#         fns = self.fns}>:
#       Package SelfM SelfF
#     end;
instantiate = <val>
            : All(SelfM:*->*)
              All(SelfI<SelfM)
              All(SelfF:*->*)
              All(SelfR)
                   SelfR
                -> (Class SelfM SelfI SelfF SelfR)
                -> (Package SelfM SelfF)
```

We can now implement a point class with an equality function. The relevant interface types are:

```
# PointR = {|x: Int|};
PointR : *
# PointM = Fun(Rep) {|setX: Rep->Int->Rep, getX: Rep->Int|};
PointM : *->*
# PointI = Fun(Rep) {|setX: Rep->Int->Rep, getX: Rep->Int, rep: Rep->PointR|};
PointI : *->*
# PointF = Fun(Obj) {|eq: Obj->Obj->Bool|};
PointF : *->*
```

The definition of a point class is just a record of methods and friendly functions, abstracted on a record `self` of methods with the same types. By also abstracting `pointClass` on a pair of functions for extracting (`get`) and overwriting (`put`), we obtain a point class that is polymorphic in the "final representation type" `FinalR`. Note that the definition below uses the `getRep` function defined in Section 4.

```
# pointClass =
#  (fun(FinalR:*)
#   fun(get: FinalR->PointR)
#   fun(put: FinalR->PointR->FinalR)
#   fun(self: PointM FinalR)
#     {methods =
#        {setX = fun(s: FinalR) fun(i: Int) put s {x=i},
#         getX = fun(s: FinalR) (get s).x,
#         rep  = fun(s: FinalR) get s},
#      fns =
#        {eq = fun(p1: Object PointI) fun(p2: Object PointI)
#            eqInt (getRep PointR p1).x (getRep PointR p2).x}
#     }): Class PointM PointI PointF PointR;
pointClass = <val> : Class PointM PointI PointF PointR
```

Point packages are created by applying `instantiate` to the appropriate interface types, a representation type, an initial value of the representation type and an appropriately typed point class:

```
# pointPackage =
#   instantiate PointM PointI PointF PointR
#   {x=5} pointClass;
pointPackage = <val> : Package PointM PointF
```

The resulting package can be manipulated in the usual way:

```
# open pointPackage as <PointI, point> in
#   let p = point.init in
#     point.fns.eq p (send_setX PointI p 1)
#   end
# end;
false : Bool
```

Finally, we can write a generic class extension function, `extend`. This function takes as parameters:

- an existing class definition `superClass`,
- a function `inc` that describes the "increment" between the methods and friendly functions 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 and friendly functions.

```
# Increment =
#  Fun(SuperM: *->*) Fun(SuperI: *->*) Fun(SuperF: *->*) % Superclass interfaces
#  Fun(NewM:   *->*) Fun(NewI:   *->*) Fun(NewF:   *->*) % New class interfaces
#  Fun(NewR) % New representation
#   All(FinalR)                        % Final representation
#     (FinalR->NewR) ->                % extractor
#     (FinalR->NewR->FinalR) ->        % overwriter
#     (SuperM FinalR) ->               % Superclass methods
#     (SuperF(Object SuperI)) ->       % Superclass friendly functions
#     (NewM FinalR) ->                 % Self methods
#     ClassResult NewI NewF FinalR; % Class result
Increment : (*->*)->(*->*)->(*->*)->(*->*)->(*->*)->(*->*)->*->*
```

The extend function is defined as follows:

```
# extend =
#   fun(SuperM: *->*)        % Superclass interface
#   fun(SuperI < SuperM)     % Superclass interface extended with rep method
#   fun(SuperF: *->*)        % Superclass friendly functions
#   fun(SuperR: *)           % Superclass internal representation type
#   fun(NewM < SuperM)       % New class interface
#   fun(NewI < NewM)         % New class interface extended with rep method
#   fun(NewF < SuperF)       % New class friendly functions
#   fun(NewR: *)             % New class internal representation type
#   fun(superClass: Class SuperM SuperI SuperF SuperR) % The superclass
#   fun(inc: Increment SuperM SuperI     % The increment function
#            SuperF NewM NewI NewF NewR)
#   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(newM: NewM FinalR)
#       let super =
#         superClass FinalR
#         (fun(s:FinalR) get(g s))
#         (fun(s:FinalR) fun(s':SuperR) p s (put (g s) s'))
#         newM
#       in inc FinalR g p super.methods super.fns newM
#       end): Class NewM NewI NewF NewR;
extend = <val>
       : All(SuperM:*->*)
         All(SuperI<SuperM)
         All(SuperF:*->*)
         All(SuperR)
         All(NewM<SuperM)
         All(NewI<NewM)
         All(NewF<SuperF)
         All(NewR)
              (Class SuperM SuperI SuperF SuperR)
```

```
          -> (Increment SuperM SuperI SuperF NewM NewI NewF NewR)
          -> (NewR->SuperR)
          -> (NewR->SuperR->NewR)
          -> (Class NewM NewI NewF NewR)
```

The class `pointClass` had no superclasses: its behavior was defined directly. Colored points, on the other hand, should be defined incrementally, by applying the `extend` function to an increment function. The relevant interface types for colored points are:

```
# CPointR = {|x: Int, c: Color|};
CPointR : *
# CPointM = Fun(Rep) {|setX: Rep->Int->Rep, getX: Rep->Int,
#                      setC: Rep->Color->Rep, getC: Rep->Color|};
CPointM : *->*
# CPointI = Fun(Rep) {|setX: Rep->Int->Rep, getX: Rep->Int,
#                      setC: Rep->Color->Rep, getC: Rep->Color,
#                      rep: Rep->CPointR|};
CPointI : *->*
# CPointF = Fun(Obj) {|eq: Obj->Obj->Bool|};
CPointF : *->*
```

The increment function maps an implementation of the point methods and friendly functions, called `super` and `superf` here, to an implementation of the colored point methods and friendly functions.

```
# cpointClass =
#   extend
#   PointM PointI PointF PointR
#   CPointM CPointI CPointF CPointR
#   pointClass
#   (fun(FinalR:*)
#    fun(get: FinalR->CPointR)
#    fun(put: FinalR->CPointR->FinalR)
#    fun(super: PointM FinalR)
#    fun(superf: PointF(Object PointI))
#    fun(self: CPointM FinalR)
#      {methods =
#         {setX = super.setX,
#          getX = super.getX,
#          setC = fun(s: FinalR) fun(newc: Color) put s {x=(get s).x, c=newc},
#          getC = fun(s: FinalR) (get s).c,
#          rep  = fun(s: FinalR) get s},
#       fns =
#         {eq = fun(p1: Object CPointI) fun(p2: Object CPointI)
#                 and
#                  (superf.eq p1 p2)
#                  (eqColor (getRep CPointR p1).c (getRep CPointR p2).c)}
#       })
#   (fun(s: CPointR) {x=s.x})
#   (fun(s: CPointR) fun(new: {|x: Int|}) {x=new.x, c=s.c});
cpointClass = <val> : Class CPointM CPointI CPointF CPointR
```

The `getX` and `setX` 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.) Note that the `eq` function of `cpointClass` is implemented in terms of a call on the the `eq` function of `pointClass`.

Instantiating `cpointClass` yields a package which can be manipulated in the usual way:

```
# cpointPackage =
#   instantiate CPointM CPointI CPointF CPointR
#   {x=5,c=red} cpointClass;
cpointPackage = <val> : Package CPointM CPointF

# open cpointPackage as <CPointI,cpoint> in
#   let p = cpoint.init in
#     cpoint.fns.eq p (send_setX CPointI p 1)
#   end
# end;
false : Bool
```

References

[BDG+88] 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.

[BM92] 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.

[Bru92] Kim B. Bruce. A paradigmatic object-oriented language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, January 1992.

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Bud91] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages.* Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[Car88a] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Preliminary version in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

[Car88b] 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.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished notes, October 1990.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. To appear in [GM93].

[Cas92] Giuseppe Castagna. Strong typing in object-oriented paradigms. Rapport de Recherche LIENS-92-11, Ecole Normale Supérieure, Paris, May 1992.

[CCH+89] 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.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM conference on LISP and Functional Programming*, pages 182–192, San Francisco, July 1992. ACM Press. Also available as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, 1992.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[dB72] Nicolas G. de Bruijn. 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, 1972.

[Ghe91] 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.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GM93] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1993. To appear.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[MHF93] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993. To appear.

[Mit90] 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. To appear in [GM93].

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[PT93] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rey85] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[Rou90a] François Rouaix. ALCOOL-90: Typage de la surcharge dans un langage fonctionnel. Thèse de doctorat, Université Paris 7, 1990.

[Rou90b] François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990.

[Rou92] François Rouaix. The Alcool–90 Report. Preliminary draft, in the Alcool distribution, April 1992.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass, 1986.