Object-Oriented Programming in Typed Lambda-Calculus: Exercises and Solutions

Benjamin C. Pierce*

June 4, 1993

Preliminaries

This document offers a series of programming exercises to accompany a recent series of research papers [HP92, PT93, PT92, Pie93b] on implementing the basic features of object-oriented programming in typed lambda-calculus. Solutions to most of the exercises are given beginning on page 10.

Most of the exercises involve programming in the typed lambda-calculus $F^{\omega}_{\leq}$. Although these exercises can be worked on paper, they will be easier and more enjoyable if solved on-line. The executable image for a prototype compiler is available by anonymous FTP:

```
ftp ftp.dcs.ed.ac.uk           [or 129.215.160.5]
login: anonymous
password: <your mail address>
cd pub/bcp
binary                         [don't forget this!]
get fomega.sun4.tar.Z
quit
uncompress fomega.sun4.tar.Z
tar xvf fomega.sun4.tar
```

The distributed binary runs only on Sun-4 processors, but it can be recompiled on any machine that supports the Standard ML of New Jersey compiler. Sources for the compiler are available from the same FTP archive, in the file `checkers.tar.Z`.

A complete description of the implementation is given in the *F-Omega-Sub User's Manual* [Pie93a], included with the distribution.

1 Polymorphism and Encapsulation

1.1. Exercise: The predefined operator `plus` forms the sum of two integer arguments:

```
> plus 2 3;
5 : Int
```

Use `plus` to compute the sum of 7, 8, 9, and 10.

1.2. Exercise: The definition

*INRIA, Projet Formel, BP 105, 78153 Le Chesnay Cedex, France. Email: `Benjamin.Pierce@inria.fr`

```
> double = fun(x:Int) plus x x;
double = <val> : Int -> Int
```

defines a new doubling function:

```
> double 5;
10 : Int
```

Use double to define a function that multiplies its argument by four.

1.3. Exercise: Records are built using curly braces:

```
> r = {a=5, b=7};
r = <val> : {|a: Int,  b: Int|}
```

The field selection operator is dot ("."):

```
> plus r.a r.b;
12 : Int
```

Write a function that takes as argument a record with two integer fields named `a` and `b` and returns a record with two fields containing the sum and product of the fields of the argument.

1.4. Exercise: The polymorphic identity function

```
> id = fun(A) fun(x:A) x;
id = <val> : All(A) A -> A
```

can be used to obtain the integer identity function by instantiating it at the type `Int`:

```
> id Int;
<val> : Int -> Int
> id Int 3;
3 : Int
```

Write a function that takes an argument of type `All(A) A->A` and (by instantiating it appropriately) applies it to itself.

1.5. Exercise: The `fomega` system includes a facility for simple type abbreviations. For example, the declaration

```
> IntFun = Int->Int;
IntFun : *
```

defines a type identifier `IntFun` that behaves identically in all contexts with the longer expression `Int->Int`:

```
> g = fun(f:IntFun) fun(x:Int) f (f x);
g = <val> : IntFun -> Int -> Int
```

Note that the type of `g` is equivalent to `IntFun->IntFun`, but this fact is not noticed by the type synthesis algorithm, which *expands* abbreviations as necessary but never tries to contract them. In situations where the abbreviated form is preferable, you can explicitly annotate parts of your program with types that you believe they possess; these assertions are verified by the typechecker. For example:

```
> g = fun(f:IntFun)
>       (fun(x:Int) f (f x))
>       :<IntFun>;
*** Warning: Old syntax: Assertions can be written e:T now ***
g = <val> : IntFun -> IntFun
```

Define an abbreviation `Id` for the type of the polymorphic identity function above. Use it to write a version of the polymorphic self-application function whose type is printed as `Id->Id`. Define another version whose type is printed as `Id->(All(A)A->A)` (there are at least two ways to do this).

1.6. Exercise: One of the interesting properties of the polymorphic lambda-calculus is that it can be used to encode many basic datatypes, such as numbers, booleans, etc. For example, the type of the "Church booleans" (c.f. [BB85, Chu41, PDM89, etc.]) is:

```
> CBool = All(A) A->A->A;
CBool : *
```

This type has two normal-form inhabitants:

```
> ctrue = fun(A) fun(t:A) fun(f:A) t;
ctrue = <val> : All(A) A -> A -> A
> cfalse = fun(A) fun(t:A) fun(f:A) f;
cfalse = <val> : All(A) A -> A -> A
```

Then, for example, the `not` function on booleans can be written

```
> cnot = fun(b:CBool) b CBool cfalse ctrue;
cnot = <val> : CBool -> CBool
```

Write a function `cand : CBool->CBool->CBool` that computes the logical "and" of its arguments. Write a three-argument logical "and" function.

1.7. Exercise: The `fomega` implementation also provides a built-in type `Bool` with the predefined constants `true` and `false` and a primitive selector `primIf : All(A) A->A->A`. The parser allows the special syntax

```
if T b then e1 else e2 end
```

as an abbreviation for

```
primIf T b e1 e2
```

Use the `if...then...else...end` syntax to define an `and` function for pairs of booleans.

1.8. Exercise: The type of pairs of integers can be defined following the model of the Church booleans:

```
> IntPair = All(A) (Int->Int->A) -> A;
IntPair : *
> pair = fun(i:Int) fun(j:Int)
>          (fun(A) fun(f:Int->Int->A)
>             f i j)
>          :<IntPair>;
*** Warning: Old syntax: Assertions can be written e:T now ***
pair = <val> : Int -> Int -> IntPair
> p = pair 3 5;
p = <val> : IntPair
> fst = fun(p:IntPair) p Int (fun(i:Int) fun(j:Int) i);
fst = <val> : IntPair -> Int
```

Define the `snd` function on pairs.

1.9. Exercise: The pervasive environment includes a polymorphic fixed point operator

```
> rec;
<val> : All(A) (A->A) -> A
```

which, given a type `A` and a function `f : A->A`, computes the "least" value `x` such that `f x = x`. Use `rec` to write a function that computes the factorial of its argument. (You will need some of the other pervasive arithmetic functions listed in the *F-Omega-Sub User's Manual*. You may also want to put your definition in a file rather than typing it directly to the interpreter. The `use` command can be used to load the file into a session with the `fomega` system, or the file may be processed by typing `fomega filename` from the unix command line.)

1.10. Exercise: The `fomega` system also supports higher-order polymorphism. For example,

```
> Pair = Fun(T:*) All(A) (T->T->A) -> A;
Pair : *->*
> pair = fun(T)
>         fun(i:T) fun(j:T)
>           (fun(A) fun(f:T->T->A)
>              f i j)
>           :<Pair(T)>;
*** Warning: Old syntax: Assertions can be written e:T now ***
pair = <val> : All(T) T -> T -> (Pair T)
> p = pair Int 3 5;
p = <val> : Pair Int
> fst = fun(T) fun(p: Pair T) p T (fun(i:T) fun(j:T) i);
fst = <val> : All(T) (Pair T) -> T
> snd = fun(T) fun(p: Pair T) p T (fun(i:T) fun(j:T) j);
snd = <val> : All(T) (Pair T) -> T
```

Extend the definition of polymorphic pairs to the case where the first and second projections may have different types.

1.11. Exercise: Consider the type:

```
> Ex = Some(A) {|e:A, f:A->Int->Int|};
Ex : *
```

A typical value of this type is

```
> p = < Int,
>       {e = 5,
>          f = fun(x:Int) fun(y:Int) x}
>     >: Ex;
p = <val> : Ex
```

Find two more expressions with this type.

1.12. Exercise: Given an element of `Ex`, the following function opens it and applies the `f` component to the `e` component:

```
> use_it = fun(p:Ex)
>           open p as <A,r> in
>             r.f r.e 2
>           end;
use_it = <val> : Ex -> Int
> use_it p;
5 : Int
```

Write a function that takes an element of `Ex` and uses it to build an element of the type `Some(A) {|e:A, f:A->Int|}`.

1.13. Exercise: How many different values for `T` can you find such that the expression

```
<Int, {a=5,b=7}> : Some(A)T
```

is well typed?

1.14. Exercise: Is there any pair of values for `T` and `T'` where `T'` is *not* `Int` and the expression

```
<T', {a=5,b=7}> : Some(A)T
```

is well typed?

1.15. Exercise: Recall the definition of the `Point` type:

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

Since we will be writing many such type expressions, it will be convenient to separate them into a common part that describes their outer structure (the existential type and the outer record) and a specific part that lists the methods and their types (the inner record). These parts will then be combined to produce specific object types such as `Point`. Define the type operator `Object:(*->*)->*` and the type `PointM:*->*` such that

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

1.16. Exercise: Use the `Object` operator and a new operator `SlidePointM` to give an extension of the type `Point` with a `slide` method of the same type as `setX`,

1.17. Exercise: In class we built a typical element of `Point` like this:

```
> p = < Int,
>       {state = 5,
>        methods = {setX = fun(x:Int) fun(i:Int) i,
>                   getX = fun(x:Int) x}}
>     >: Point ;
p = <val> : Point
```

Show how to build a typical element of `Object(SlidePointM)`.

1.18. Exercise: Recall the definition of the `Point'setX` function, which "sends the message `setX`" to elements of `Object(PointM)`:

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

The definition of `SlidePoint'setX` is identical except for the type of the first parameter. Define `SlidePoint'slide` and `SlidePoint'getX`.

1.19. Exercise: Here is an interface specification for colored points:

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

Write a function that takes an element of `Object(PointM)` and uses it to build an element of `Object(CPointM)`.

2 Subtyping

2.1. Exercise: Use the inference rules defining the $F^\omega_\leq$ subtype relation to decide how the `fomega` implementation will respond to the following assertions.

```
check Top(*) < Top(*);
check Int < Top(*);
check Top(*) < Int;

check {|a:Int, b:String|} < {|b:String|};
check {|a:Int, b:String|} < {|a:Bool, b:String|};
check {|a:Int, b:String|} < {|a:Int, c:String|};
check {|a:Int, b:String|} < {|a:Int, b:Top(*)|};

Pos < Int;     % (Declare a new type variable Pos with bound Int)
check Pos < Int;
check Int < Pos;
check Int->Pos < Int->Int;
check Int->Int < Pos->Pos;
check Int->Pos < Pos->Int;
check Pos->Int < Int->Pos;

check All(A)A->A < Int->Int;
check All(B)B < All(A)A;
check All(B)B->Pos < All(A)A->Int;
check All(A<Int)A->A < All(A<Pos)A->Pos;
check Some(A<Pos) {|x:A, y:A->Pos|}  <  Some(B<Int) {|x:Int, y:B->Pos|};
check Some(A<Int) {|x:A, y:A->A|}  <  Some(A<Int) {|x:Int, y:A->Pos|};

check (Fun(A:*) A->A) Pos  <  Pos->Int;
check (Fun(A:*) A->A) Pos  <  (Fun(B:*) Pos->B) Int;

check Fun(A:*) Pos->A  <  Top(*->*);
check Fun(A:*) {|x:A, y:A->A|}  <  Fun(A:*) {|y:A->A|};
check Fun(A:*) All(B<A) A->A  <  Fun(A:*) All(B<Top(*)) B->A;

check All(M < Fun(A:*) {|x:A,y:Int|}) M(Pos)
    < All(M < Fun(A:*) {|x:A,y:Int|}) {|x:Int|};
check All(M < Fun(A:*) {|x:A,y:Int|}) M(Pos)
    <  All(M < Fun(A:*) {|x:A,y:Int|}) M(Int);
check All(M < Fun(A:*) {|x:A,y:Int|}) M(Pos)
    <  All(M < Fun(A:*) {|y:Int|}) M(Int);

check Top(*) < Top(*->*);
```

2.2. Exercise: In Exercise 1.13, we found four different values for `T` such that

```
<Int, {a=5,b=7}> : T
```

is well typed. Using subtyping and bounded existential quantifiers, how many more can you find?

2.3. Exercise: Convince yourself that the first version we tried of the function for "sending the `setX` message to a point object"

```
wrong'setX =
  fun(P<Object(PointM))
    fun(p:P)
      open p as <Rep,r> in
         fun(i:Int)
            <Rep, {state = r.methods.setX r.state i,
                   methods = r.methods}
           > : P
      end;
```

is really ill typed according to the typing rules of $F^{\omega}_{\leq}$.

2.4. Exercise: Extend your implementation of slideable points from Exercises 1.16 and 1.18 so that the functions `SlidePoint'setX`, `SlidePoint'getX`, and `SlidePoint'slide` deal properly with subtyping.

```
SlidePointM : *->*
```

2.5. Exercise: Give a type operator `SlideCPointM` capturing the interface specification of slideable colored points. Define the functions `SlideCPoint'setC` and `SlideCPoint'getC` and build a typical element of `Object(SlideCPointM)`. Write a small program that uses your functions to manipulate this object by sending it messages.

2.6. Exercise: Note that we can also define the functions `SlideCPoint'setX`, `SlideCPoint'getX`, and `SlideCPoint'slide`. Does their behavior differ from the corresponding methods for slideable points? Is it useful to have both sets of functions? (If not, can you think of any circumstances where it would be useful?)

2.7. Exercise: (Thought exercise) We have now developed a full account of the two most basic features of object-oriented programming: encapsulation and subtyping. Moreover, we have seen (e.g. in Exercise 1.19) some ways of building one object from another using much simpler mechanisms than full object-oriented inheritance. It might now be argued that we have already captured most of what is meant by "conventional object-oriented programming" — that we (as language designers) should ignore the complexities of inheritance and simply invent a little bit of high-level concrete syntax to make it more convenient to program with the mechanisms we have developed so far. Would you agree with this proposition?

3 Inheritance

3.1. Exercise: The file `simpinh.f` (listed in Figure 1 and available as part of the `fomega` distribution) contains a simple implementation of inheritance (without the ability to "send messages to `self`"). Copy the file `simpinh.f` and add a simple `slide` method to the definitions of points; this `slide` should be inherited by the definition of colored points. For the moment, you will not be able to get exactly the same behavior as in the Smalltalk implementation of slideable points, where the `slide` method defined by the point class calls the overriden `setX` method of colored points. Define `slide` here so that it sets the instance variable `x` directly.

3.2. Exercise: Adding `self` is actually quite easy. Each class is extended with an extra parameter for the "`self` methods. For example, the `pointClass` from the lecture becomes:

```
> 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}
>      } :<PointM FinalR>;
*** Warning: Old syntax: Assertions can be written e:T now ***
pointClass = <val>
           : All(FinalR)
                 (FinalR->PointR)
              -> (FinalR->PointR->FinalR)
              -> (PointM FinalR)
              -> (PointM FinalR)
```

Then building a new object from a class involves using a fixed point to build the actual method vector:

```
Class =
  Fun(M:*->*)
  Fun(R:*)
    All(FinalR)
         (FinalR->R)
      -> (FinalR->R->FinalR)
      -> M(FinalR);

new =
  fun(M<Top(*->*))
  fun(R)
  fun(s:R)
  fun(cl: Class M R)
    < R,
      {state = s,
       methods = cl
                   R
                   (fun(s:R) s)
                   (fun(s:R) fun(s':R) s')}
    >: Object(M);

extend =
  fun(SuperM < Top(*->*))                      % superclass interface
  fun(SuperR)                                  % superclass representation
  fun(NewM < SuperM)                           % new class interface
  fun(NewR)                                    % new class representation
  fun(superClass: Class SuperM SuperR)         % the superclass
  fun(build:                                   % "increment" fun, with args...
          All(FinalR)                             % final representation
          (FinalR->NewR) ->                       % extractor
          (FinalR->NewR->FinalR) ->               % overwriter
          (SuperM(FinalR)) ->                     % superclass methods
          (NewM(FinalR)))                         % ...returning new methods
  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)
      build FinalR g p
         (superClass FinalR
           (fun(s:FinalR) get(g(s)))
           (fun(s:FinalR) fun(s':SuperR) p s (put (g s) s'))));

PointM = Fun(R:*) {|setX:R->Int->R, getX:R->Int|};
PointR = {|x:Int|};
PointClass = Class PointM PointR;

CPointM = Fun(R:*) {|setX:R->Int->R, getX:R->Int,
                     setC:R->Color->R, getC:R->Color|};
CPointR = {|x:Int, color:Color|};
CPointClass = Class CPointM CPointR;

p_class =
   fun(FinalR)
   fun(get: FinalR->PointR)
   fun(put: FinalR->PointR->FinalR)
     {setX = fun(s:FinalR) fun(i:Int) put s {x=i},
      getX = fun(s:FinalR) (get s).x};

cp_class =
  extend PointM PointR CPointM CPointR p_class
  (fun(FinalR)
   fun(get: FinalR->CPointR)
   fun(put: FinalR->CPointR->FinalR)
   fun(super: PointM(FinalR))
    {setX = super.setX,
     getX = super.getX,
     setC = fun(s:FinalR) fun(c:Color)
               put s {x=(get s).x, color=c},
     getC = fun(s:FinalR) (get s).color})
  (fun(s:CPointR) {x=s.x})
  (fun(s:CPointR) fun(s':PointR) {x=s'.x, color=s.color});

c = new CPointM CPointR {x=1,color=red} cp_class;
```

Figure 1: The file `simpinh.f`

```
> p = < PointR,
>       {state = {x=5},
>        methods = rec (PointM(PointR))
>                    fun(self:PointM(PointR))
>                      pointClass
>                        PointR
>                        (fun(s:PointR) s)
>                        (fun(s:PointR) fun(s':PointR) s')
>                        self}
>     >: Object(PointM);
p = <val> : Object PointM
```

Extend this example with a `slide` method that uses the `setX` and `getX` methods from `self`.

3.3. Exercise: By analogy with the previous exercise, show how to add `self` to the definitions of `Class`, `new`, and `extend` in the file `simpinh.f`.

3.4. Exercise: Use `extend` to give a slideable version of the colored point class that inherits the slide method from slidePointClass. Show that instances of this class have the same behavior as their Smalltalk counterparts (i.e. the `slide` method inherited from points uses the overriden `setX` method of colored points).

Acknowledgements

Many thanks to the participants in the 1992 Frankische OOrientierungstage, on whom earlier versions of these exercises were first inflicted, and to Adriana Compagnoni, who corrected a number of mistakes in a later version.

This research was partially supported by ESPRIT Basic Research Action "TYPES."

Solutions

1 Polymorphism and Encapsulation

1.1. Exercise: *The predefined operator* `plus` *forms the sum of two integer arguments:*

```
> plus 2 3;
5 : Int
```

Use `plus` *to compute the sum of 7, 8, 9, and 10.*
One possibility:

```
> plus (plus 7 8) (plus 9 10);
34 : Int
```

1.2. Exercise: *The definition*

```
> double = fun(x:Int) plus x x;
double = <val> : Int -> Int
```

defines a new doubling function:

```
> double 5;
10 : Int
```

Use double to define a function that multiplies its argument by four.

```
> quad = fun(x:Int) double (double x);
quad = <val> : Int -> Int
```

1.3. Exercise: *Records are built using curly braces:*

```
> r = {a=5, b=7};
r = <val> : {|a: Int,  b: Int|}
```

The field selection operator is dot ("."):

```
> plus r.a r.b;
12 : Int
```

Write a function that takes as argument a record with two integer fields named `a` *and* `b` *and returns a record with two fields containing the sum and product of the fields of the argument.*

```
> f = fun(r:{a:Int,b:Int})
>       {sum = plus r.a r.b,
>        product = times r.a r.b};
*** Warning: Old syntax: Use {| and |} for record types ***
f = <val> : {|a:Int, b:Int|} -> {|sum:Int, product:Int|}
> f r;
<val> : {|sum: Int,  product: Int|}
```

1.4. Exercise: *The polymorphic identity function*

```
> id = fun(A) fun(x:A) x;
id = <val> : All(A) A -> A
```

can be used to obtain the integer identity function by instantiating it at the type `Int`:

```
> id Int;
<val> : Int -> Int
> id Int 3;
3 : Int
```

Write a function that takes an argument of type `All(A) A->A` and (by instantiating it appropriately) applies it to itself.

```
> delta = fun(f:All(A) A->A) f (All(A) A->A) f;
delta = <val> : (All(A)A->A) -> (All(A)A->A)
> delta id;
<val> : All(A) A -> A
```

1.5. Exercise: *Define an abbreviation `Id` for the type of the polymorphic identity function above. Use it to write a version of the polymorphic self-application function whose type is printed as `Id->Id`. Define another version whose type is printed as `Id->(All(A)A->A)` (there are at least two ways to do this).*

```
> Id = All(A) A->A;
Id : *
> delta = fun(x:Id) x Id x;
delta = <val> : Id -> Id
> delta' = fun(x:Id) x (All(A)A->A) x;
delta' = <val> : Id -> (All(A)A->A)
> delta'' = fun(x:Id) (x Id x) :<All(A)A->A>;
*** Warning: Old syntax: Assertions can be written e:T now ***
delta'' = <val> : Id -> (All(A)A->A)
```

1.6. Exercise: *Write a function `cand : CBool->CBool->CBool` that computes the logical "and" of its arguments. Write a three-argument logical "and" function.*

```
> cand = fun(a:CBool) fun(b:CBool)
>          a CBool b cfalse
>          :<CBool>;
*** Warning: Old syntax: Assertions can be written e:T now ***
cand = <val> : CBool -> CBool -> CBool
> cand ctrue cfalse;
<val> : CBool
> cand3 = fun(a:CBool) fun(b:CBool) fun(c:CBool)
>          a CBool (b CBool c cfalse) cfalse
>          :<CBool>;
*** Warning: Old syntax: Assertions can be written e:T now ***
cand3 = <val> : CBool -> CBool -> CBool -> CBool
> cand3 ctrue ctrue ctrue;
<val> : CBool
```

1.7. Exercise: *Use the `if...then...else...end` syntax to define an `and` function for pairs of booleans.*

```
> and = fun(a:Bool) fun(b:Bool)
>         if Bool a
>            then b
>            else false
>         end;
and = <val> : Bool -> Bool -> Bool
> and true false;
false : Bool
```

1.8. Exercise: *Define the `snd` function on pairs.*

```
> snd = fun(p:IntPair) p Int (fun(i:Int) fun(j:Int) j);
snd = <val> : IntPair -> Int
> times (fst p) (snd p);
15 : Int
```

1.9. Exercise: *Use `rec` to write a function that computes the factorial of its argument.*

```
> fact =
>   rec (Int->Int)
>      fun(f:Int->Int)
>        fun(n:Int)
>          if Int eqInt n 0
>            then 1
>            else times n (f (minus n 1))
>          end;
fact = <val> : Int -> Int
> fact 6;
720 : Int
```

1.10. Exercise: *Extend the definition of polymorphic pairs to the case where the first and second projections may have different types.*

```
> Pair = Fun(S:*) Fun(T:*) All(A) (S->T->A) -> A;
Pair : *->*->*
> pair = fun(S) fun(T)
>          fun(i:S) fun(j:T)
>            (fun(A) fun(f:S->T->A)
>               f i j)
>            :<Pair S T>;
*** Warning: Old syntax: Assertions can be written e:T now ***
pair = <val> : All(S) All(T) S -> T -> (Pair S T)
> p = pair Bool Int true 5;
p = <val> : Pair Bool Int
> fst = fun(S) fun(T) fun(p: Pair S T) p S (fun(i:S) fun(j:T) i);
fst = <val> : All(S) All(T) (Pair S T) -> S
> snd = fun(S) fun(T) fun(p: Pair S T) p T (fun(i:S) fun(j:T) j);
snd = <val> : All(S) All(T) (Pair S T) -> T
```

1.11. Exercise: *Consider the type:*

```
> Ex = Some(A) {|e:A, f:A->Int->Int|};
Ex : *
```

A typical value of this type is

```
> p = < Int,
>       {e = 5,
>            f = fun(x:Int) fun(y:Int) x}
>     >: Ex;
p = <val> : Ex
```

Find two more expressions with this type.

For example:

```
> <Int, {e = 5, f = fun(x:Int) fun(y:Int) plus x y}> : Ex;
<val> : Ex
> <Bool, {e = true, f = fun(x:Bool) fun(y:Int) y}> : Ex;
<val> : Ex
```

1.12. Exercise: *Write a function that takes an element of `Ex` and uses it to build an element of the type `Some(A) {|e:A, f:A->Int|}`.*

```
> fun(p:Ex)
>   open p as <A,r> in
>     < A,
>       {e = r.e,
>        f = fun(a:A) r.f a 5}
>     >: Some(A) {|e:A, f:A->Int|}
>   end;
<val> : Ex -> (Some(A){|e:A, f:A->Int|})
```

1.13. Exercise: *How many different values for `T` can you find such that the expression*

`<Int, {a=5,b=7}> : Some(A)T`

is well typed?

Here are all the ones I can think of:

```
> <Int, {a=5,b=7}> : Some(A){|a:A,b:A|};
<val> : Some(A) {|a: A,  b: A|}
> <Int, {a=5,b=7}> : Some(A){|a:Int,b:A|};
<val> : Some(A) {|a: Int,  b: A|}
> <Int, {a=5,b=7}> : Some(A){|a:A,b:Int|};
<val> : Some(A) {|a: A,  b: Int|}
> <Int, {a=5,b=7}> : Some(A){|a:Int,b:Int|};
<val> : Some(A) {|a: Int,  b: Int|}
```

1.14. Exercise: *Is there any pair of values for `T` and `T'` where `T'` is* not *`Int` and the expression*

`<T', {a=5,b=7}> : Some(A)T`

is well typed?

Yes, infinitely many. For example:

```
> <Bool, {a=5,b=7}> : Some(A){|a:Int,b:Int|};
<val> : Some(A) {|a: Int,  b: Int|}
```

1.15. Exercise: *Recall the definition of the `Point` type:*

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

Define the type operator `Object:(*->*)->*` and the type `PointM:*->*` such that

`Point = Object(PointM)`

```
> Object = Fun(M:*->*) Some(R) {|state: R, methods: M(R)|};
Object : (*->*)->*
> PointM = Fun(R) {|setX: R->Int->R, getX: R->Int|};
PointM : *->*
> check Point ~ Object(PointM);
Yes
```

1.16. Exercise: *Use the `Object` operator and a new operator `SlidePointM` to give an extension of the type `Point` with a `slide` method of the same type as `setX`,*

```
> SlidePointM = Fun(R) {|setX: R->Int->R, getX: R->Int, slide: R->Int->R|};
SlidePointM : *->*
> SlidePoint = Object(SlidePointM);
SlidePoint : *
```

1.17. Exercise: *Show how to build a typical element of `Object(SlidePointM)`.*

```
> mp = < Int,
>        {state = 5,
>         methods = {setX = fun(x:Int) fun(i:Int) i,
>                    slide = fun(x:Int) fun(i:Int) plus x i,
>                    getX = fun(x:Int) x}}
>     >: Object(SlidePointM) ;
mp = <val> : Object SlidePointM
```

1.18. Exercise: *Define `SlidePoint'slide` and `SlidePoint'getX`.*

```
> SlidePoint'slide =
>   fun(p:Object(SlidePointM))
>     fun(i:Int)
>       open p as <R,r> in
>         < R,
>           {state = r.methods.slide r.state i,
>            methods = r.methods}
>         >: Object(SlidePointM)
>        end;
SlidePoint'slide = <val>
                 : (Object SlidePointM) -> Int -> (Object SlidePointM)
> SlidePoint'getX =
>   fun(p:Object(SlidePointM))
>       open p as <R,r> in
>         r.methods.getX r.state
>       end;
SlidePoint'getX = <val> : (Object SlidePointM) -> Int
```

1.19. Exercise: *Write a function that takes an element of `Object(PointM)` and uses it to build an element of `Object(CPointM)`.*

Here are two possibilities:

```
> cpoint_from_point =
>   fun(p:Object(PointM))
>     < {| pstate:Object(PointM), cstate:Color |},
>       {state = {pstate = p,
>                 cstate = red},
>        methods =
>           {setX = fun(s:{|pstate:Object(PointM),cstate:Color|}) fun(i:Int)
>                     let new_pstate = Point'setX s.pstate i
>                     in {pstate=new_pstate, cstate=s.cstate}
>                     end,
>            getX = fun(s:{|pstate:Object(PointM),cstate:Color|})
>                     Point'getX s.pstate,
>            setC = fun(s:{|pstate:Object(PointM),cstate:Color|}) fun(c:Color)
>                     {pstate=s.pstate, cstate=c},
```

```
>           getC = fun(s:{|pstate:Object(PointM),cstate:Color|})
>                    s.cstate}}
>     >: Object(CPointM);
cpoint_from_point = <val> : (Object PointM) -> (Object CPointM)

> cpoint_from_point' =
>   fun(p:Object(PointM))
>     open p as <R,r> in
>       < {|pstate:R, cstate:Color|},
>         {state = {pstate = r.state,
>                   cstate = red},
>          methods =
>             {setX = fun(s:{|pstate:R,cstate:Color|}) fun(i:Int)
>                        let new_pstate = r.methods.setX s.pstate i
>                        in {pstate=new_pstate, cstate=s.cstate}
>                        end,
>              getX = fun(s:{|pstate:R,cstate:Color|})
>                        r.methods.getX s.pstate,
>              setC = fun(s:{|pstate:R,cstate:Color|}) fun(c:Color)
>                        {pstate=s.pstate, cstate=c},
>              getC = fun(s:{|pstate:R,cstate:Color|})
>                        s.cstate}}
>      >: Object(CPointM)
>     end;
cpoint_from_point' = <val> : (Object PointM) -> (Object CPointM)
```

2 Subtyping

2.1. Exercise: *Use the inference rules defining the $F^{\omega}_{\leq}$ subtype relation to decide how the* **fomega** *implementation will respond to the following assertions.*

```
> check Top(*) < Top(*);
Yes
> check Int < Top(*);
Yes
> check Top(*) < Int;
No

> check {|a:Int, b:String|} < {|b:String|};
Yes
> check {|a:Int, b:String|} < {|a:Bool, b:String|};
No
> check {|a:Int, b:String|} < {|a:Int, c:String|};
No
> check {|a:Int, b:String|} < {|a:Int, b:Top(*)|};
Yes

> Pos < Int;     % (Declare a new type variable Pos with bound Int)
> check Pos < Int;
Yes
> check Int < Pos;
No
> check Int->Pos < Int->Int;
Yes
> check Int->Int < Pos->Pos;
No
> check Int->Pos < Pos->Int;
```

```
Yes
> check Pos->Int < Int->Pos;
No

> check All(A)A->A < Int->Int;
No
> check All(B)B < All(A)A;
Yes
> check All(B)B->Pos < All(A)A->Int;
Yes
> check All(A<Int)A->A < All(A<Pos)A->Pos;
Yes
> check Some(A<Pos) {|x:A, y:A->Pos|}  <  Some(B<Int) {|x:Int, y:B->Pos|};
Yes
> check Some(A<Int) {|x:A, y:A->A|}  <  Some(A<Int) {|x:Int, y:A->Pos|};
No

> check (Fun(A:*) A->A) Pos  <  Pos->Int;
Yes
> check (Fun(A:*) A->A) Pos  <  (Fun(B:*) Pos->B) Int;
Yes

> check Fun(A:*) Pos->A  <  Top(*->*);
Yes
> check Fun(A:*) {|x:A, y:A->A|}  <  Fun(A:*) {|y:A->A|};
Yes
> check Fun(A:*) All(B<A) A->A  <  Fun(A:*) All(B<Top(*)) B->A;
No

> check All(M < Fun(A:*) {|x:A,y:Int|}) M(Pos)
>     < All(M < Fun(A:*) {|x:A,y:Int|}) {|x:Int|};
Yes
> check All(M < Fun(A:*) {|x:A,y:Int|}) M(Pos)
>     <  All(M < Fun(A:*) {|x:A,y:Int|}) M(Int);
No
> check All(M < Fun(A:*) {|x:A,y:Int|}) M(Pos)
>     <  All(M < Fun(A:*) {|y:Int|}) M(Int);
No

> check Top(*) < Top(*->*);
No
```

2.2. Exercise: *In Exercise 1.13, we found four different values for* T *such that*

    <Int, {a=5,b=7}> : T

is well typed. Using subtyping and bounded existential quantifiers, how many more can you find?

Here are a few:

```
> <Int, {a=5,b=7}> : Some(A){|a:A|};
<val> : Some(A) {|a: A|}
> <Int, {a=5,b=7}> : Some(A){|a:A,b:Top(*)|};
<val> : Some(A) {|a: A,  b: Top(*)|}
> <Int, {a=5,b=7}> : Some(A)Top(*);
<val> : Some(A) Top(*)
> <Int, {a=5,b=7}> : Some(A<Int){|a:A,b:A|};
<val> : Some(A<Int) {|a: A,  b: A|}
```

```
> <Int, {a=5,b=7}> : Some(A<Int){|a:A|};
<val> : Some(A<Int) {|a: A|}
```

2.3. Exercise: *Convince yourself that the first version we tried of the function for "sending the* `setX` *message to a point object"*

```
wrong'setX =
  fun(P<Object(PointM))
    fun(p:P)
      open p as <Rep,r> in
        fun(i:Int)
          <Rep, {state = r.methods.setX r.state i,
                 methods = r.methods}
          > : P
      end;
```

is really ill typed according to the typing rules of $F^{\omega}_{\leq}$*.*

2.4. Exercise: *Extend your implementation of slideable points from Exercises 1.16 and 1.18 so that the functions* `SlidePoint'setX`*,* `SlidePoint'getX`*, and* `SlidePoint'slide` *deal properly with subtyping.*

```
SlidePointM : *->*

> SlidePoint'setX =
>  fun(M<SlidePointM)
>    fun(p:Object(M))
>      fun(i:Int)
>        open p as <R,r> in
>          <R,
>           {state = r.methods.setX r.state i,
>            methods = r.methods}
>          >: Object(M)
>       end;
SlidePoint'setX = <val> : All(M<SlidePointM) (Object M) -> Int -> (Object M)
> SlidePoint'slide =
>  fun(M<SlidePointM)
>    fun(p:Object(M))
>      fun(i:Int)
>        open p as <R,r> in
>          <R,
>           {state = r.methods.slide r.state i,
>            methods = r.methods}
>          >: Object(M)
>      end;
SlidePoint'slide = <val>
                 : All(M<SlidePointM) (Object M) -> Int -> (Object M)
> SlidePoint'getX =
>  fun(M<SlidePointM)
>    fun(p:Object(M))
>       open p as <R,r> in
>         r.methods.getX r.state
>       end;
SlidePoint'getX = <val> : All(M<SlidePointM) (Object M) -> Int
```

2.5. Exercise: *Give a type operator* `SlideCPointM` *capturing the interface specification of slideable colored points. Define the functions* `SlideCPoint'setC` *and* `SlideCPoint'getC` *and build a typical element of* `Object(SlideCPointM)`*. Write a small program that uses your functions to manipulate this object by sending it messages.*

```
> SlideCPointM = Fun(R) {|setX: R->Int->R, getX: R->Int, slide: R->Int->R,
>                         setC: R->Color->R, getC: R->Color|};
SlideCPointM : *->*

> SlideCPoint'setC =
>  fun(M<SlideCPointM)
>    fun(p:Object(M))
>      fun(c:Color)
>        open p as <R,r> in
>          <R,
>           {state = r.methods.setC r.state c,
>            methods = r.methods}
>          >: Object(M)
>       end;
SlideCPoint'setC = <val>
                 : All(M<SlideCPointM) (Object M) -> Color -> (Object M)

> SlideCPoint'getC =
>  fun(M<SlideCPointM)
>    fun(p:Object(M))
>       open p as <R,r> in
>         r.methods.getC r.state
>       end;
SlideCPoint'getC = <val> : All(M<SlideCPointM) (Object M) -> Color

> c = < {|x:Int,color:Color|},
>       {state = {x=5,color=red},
>        methods = {setX = fun(s:{|x:Int,color:Color|}) fun(i:Int)
>                          {x=i,color=s.color},
>                   getX = fun(s:{|x:Int,color:Color|})
>                          s.x,
>                   slide = fun(s:{|x:Int,color:Color|}) fun(i:Int)
>                          {x=plus s.x i, color=blue},
>                   setC = fun(s:{|x:Int,color:Color|}) fun(c:Color)
>                          {x=s.x,color=c},
>                   getC = fun(s:{|x:Int,color:Color|})
>                          s.color
>                  }}
>      > : Object(SlideCPointM);
c = <val> : Object SlideCPointM

> let c1 = SlidePoint'setX SlideCPointM c 3 in
> let c2 = SlideCPoint'setC SlideCPointM c1 yellow in
> let c3 = SlidePoint'slide SlideCPointM c2 3 in
>   SlideCPoint'getC SlideCPointM c3
> end end end;
blue : Color
```

Note that the `slide` method behaves the same as in the Smalltalk example, but that this behavior is achieved "directly" in the implementation of `slide`, not by inheritance.

2.6. Exercise: *Note that we can also define the functions `SlideCPoint'setX`, `SlideCPoint'getX`, and `SlideCPoint'slide`. Does their behavior differ from the corresponding methods for slideable points? Is it useful to have both sets of functions? (If not, can you think of any circumstances where it would be useful?)*

No, in situations where both

```
SlidePoint'setX M p i
```

and

```
SlideCPoint'setX M p i
```

are well typed, their behavior is exactly the same.

It will always be the case that the different versions of a message sending function will behave identically whenever their behavior can be compared. (In fact, this observation might be taken as part of the *definition* of the operation of sending a message; see [HP92].) Nevertheless, it may still be useful to have the different versions available, since the type returned by the operation of sending a particular message may sometimes be refined by subclasses. For example, the `getX` method of instances of `SlideCPoint` might be defined so that it returned a `Real` instead of an `Int`.

2.7. Exercise: *(Thought exercise) We have now developed a full account of the two most basic features of object-oriented programming: encapsulation and subtyping. Moreover, we have seen (e.g. in Exercise 1.19) some ways of building one object from another using much simpler mechanisms than full object-oriented inheritance. It might now be argued that we have already captured most of what is meant by "conventional object-oriented programming" — that we (as language designers) should ignore the complexities of inheritance and simply invent a little bit of high-level concrete syntax to make it more convenient to program with the mechanisms we have developed so far. Would you agree with this proposition?*

Personally, I'm not sure whether I agree with it, but it probably ought to be tested.

3 Inheritance

3.1. Exercise: *Copy the file `simpinh.f` and add a simple `slide` method to the definitions of points; this `slide` should be inherited by the definition of colored points. For the moment, you will not be able to get exactly the same behavior as in the Smalltalk implementation of slideable points, where the `slide` method defined by the point class calls the overriden `setX` method of colored points. Define `slide` here so that it sets the instance variable x directly.*

Should be easy.

3.2. Exercise: *Extend this example with a `slide` method that uses the `setX` and `getX` methods from `self`.*

```
> SlidePointM = Fun(R:*) {|setX:R->Int->R, getX:R->Int, slide:R->Int->R|};
SlidePointM : *->*
> slidePointClass =
>    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(i:Int)
>                  self.setX s (plus (self.getX s) i)
>      } :<SlidePointM FinalR>;
*** Warning: Old syntax: Assertions can be written e:T now ***
slidePointClass = <val>
                : All(FinalR)
                       (FinalR->PointR)
                    -> (FinalR->PointR->FinalR)
                    -> (PointM FinalR)
                    -> (SlidePointM FinalR)
```

3.3. Exercise: *By analogy with the previous exercise, show how to add `self` to the definitions of `Class`, `new`, and `extend` in the file `simpinh.f`.*

```
> Class =
>   Fun(M:*->*)
>   Fun(R:*)
>     All(FinalR)
>          (FinalR->R)
>       -> (FinalR->R->FinalR)
>       -> M(FinalR)
>       -> M(FinalR);
Class : (*->*)->*->*
>
> new =
>   fun(M<Top(*->*))
>   fun(R)
>   fun(s:R)
>   fun(cl: Class M R)
>     < R,
>       {state = s,
>        methods = rec (M(R))
>                    fun(self:M(R))
>                       cl
>                         R
>                         (fun(s:R) s)
>                         (fun(s:R) fun(s':R) s')
>                         self}
>     >: Object(M);
new = <val> : All(M:*->*) All(R) R -> (Class M R) -> (Object M)
>
> extend =
>   fun(SuperM < Top(*->*))                    % superclass interface
>   fun(SuperR)                                % superclass representation
>   fun(NewM < SuperM)                         % new class interface
>   fun(NewR)                                  % new class representation
>   fun(superClass: Class SuperM SuperR)       % the superclass
>   fun(build:                                 % "increment" fun, with args...
>          All(FinalR)                            % final representation
>          (FinalR->NewR) ->                      % extractor
>          (FinalR->NewR->FinalR) ->              % overwriter
>          (SuperM(FinalR)) ->                    % superclass methods
>          (NewM(FinalR)) ->                      % self methods
>          (NewM(FinalR)))                        % ...returning new methods
>   fun(get: NewR->SuperR)                     % new->super extractor
>   fun(put: NewR->SuperR->NewR)               % new<-super overwriter
>    % Build the extended class...
>    (fun(FinalR)
>     fun(g: FinalR->NewR)
>     fun(p: FinalR->NewR->FinalR)
>     fun(self: NewM(FinalR))
>       build 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);
extend = <val>
       : All(SuperM:*->*)
         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)
       -> (All(FinalR)
               (FinalR->NewR)
             ->(FinalR->NewR->FinalR)
             ->(NewM FinalR)
             ->(NewM FinalR))
```

3.4. Exercise: *Use* `extend` *to give a slideable version of the colored point class that inherits the slide method from slidePointClass. Show that instances of this class have the same behavior as their Smalltalk counterparts (i.e. the* `slide` *method inherited from points uses the overriden* `setX` *method of colored points).*

```
> SlideCPointM = Fun(R:*) {|setX:R->Int->R, getX:R->Int,
>                           setC:R->Color->R, getC:R->Color,
>                           slide:R->Int->R|};
SlideCPointM : *->*
> CPointR = {|x:Int, color:Color|};
CPointR : *

> slideCPointClass =
>   extend SlidePointM PointR SlideCPointM CPointR slidePointClass
>   (fun(FinalR)
>    fun(get: FinalR->CPointR)
>    fun(put: FinalR->CPointR->FinalR)
>    fun(super: SlidePointM(FinalR))
>    fun(self: SlideCPointM(FinalR))
>     {setX = fun(s:FinalR) fun(i:Int)
>               put s {x=i,color=blue},
>      getX = super.getX,
>      slide = super.slide,
>      setC = fun(s:FinalR) fun(c:Color)
>                put s {x=(get s).x, color=c},
>      getC = fun(s:FinalR) (get s).color})
>   (fun(s:CPointR) {x=s.x})
>   (fun(s:CPointR) fun(s':PointR) {x=s'.x, color=s.color});
slideCPointClass = <val>
                : All(FinalR)
                       (FinalR->CPointR)
                    -> (FinalR->CPointR->FinalR)
                    -> (SlideCPointM FinalR)
                    -> (SlideCPointM FinalR)
```

After loading the functions for sending messages to points), we can test this implementation as before:

```
> c = new SlideCPointM CPointR {x=5,color=red} slideCPointClass;
c = <val> : Object SlideCPointM
> c1 = SlidePoint'setX SlideCPointM c 3;
c1 = <val> : Object SlideCPointM
> c2 = SlideCPoint'setC SlideCPointM c1 yellow;
c2 = <val> : Object SlideCPointM
> c3 = SlidePoint'slide SlideCPointM c2 3;
```

```
c3 = <val> : Object SlideCPointM
> SlideCPoint'getC SlideCPointM c3;
blue : Color
```

References

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[HP92] 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.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989.

[Pie93a] Benjamin C. Pierce. F-omega-sub user's manual, version 1.4. Available by FTP as part of the `fomega` implementation, Feb 1993.

[Pie93b] Benjamin C. Pierce. A model of delegation based on existential types. Unpublished note, April 1993.

[PT92] Benjamin C. Pierce and David N. Turner. Statically typed multi-methods via partially abstract types. Draft technical teport, September 1992.

[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".