Typed Foundations of Object-oriented Programming

POPL '92 Tutorial

Luca Cardelli
DEC SRC, 130 Lytton Avenue, Palo Alto CA 94301
luca@src.dec.com

Outline

- Basic notions and puzzles.
- Back to foundations.
- Forward to objects.

○ Approach: take a (deceivingly) simple o-o program and try to express it in "typed λ-calculus". Or, more precisely: desperately look for *any* typed λ-calculus that can express such a program.

○ Two main threads:
- Subtyping for its own sake.
- Subtyping vs. inheritance.

○ One main bias: extensible records.

Part 1. Basic notions and puzzles.

Basic notions and first modeling attempts.

What can subtyping say about o-o concepts?

What can subtyping achieve on its own?

O-o languages features

Object-oriented programming bundles together a number of important concepts, including:

Modularization	(via class signatures)
Abstraction	(via the method discipline)
Extensibility	(via subclasses and inheritance)

But the characterizing property is *extensibility*: reusing and extending existing code **without editing** it.

These properties are achieved in large part by extending vanilla procedural languages with:

(1) *Subtyping*
(*f(a)* is ok if *a* is **good enough** for *f*)

(2) *Inheritance*
(*self*, and its amazing type rules)

On the road to o-o

v-p: vanilla-procedural	(Algol,	Modula-2,	C)
s-e: subtype-enriched	⇓	⇓	⇓
o-o: object-oriented	(Simula,	Modula-3,	C++)

How much complexity is added by the first step? How much by the second? We want to know because:

(1) O-o languages have a surprisingly difficult semantics (and program logic). Moreover,
(2) they have a surprisingly difficult type theory.

We would like to understand them better. For (1) we can apply well-established semantic techniques; e.g. untyped λ-calculi (den.sem.) or to Hoare logics.
For (2) we need something much less well-established: a sufficiently expressive typed calculus.

Subtyping without inheritance

What happens if we add subtyping to a v-p language, but *not* inheritance? We do not get o-o programming (according to most definitions), but:

- This is an important stepping stone in understanding the more complex structure of full o-o languages.
- It helps making clear what inheritance really contributes, both in terms of complexity and usefulness.
- The s-e language paradigm is worth investigating on its own. It is distinct from both v-p and o-o. In some dimensions it is richer than o-o. Has some of the advantages of o-o and lacks some of its disadvantages.
- We concentrate on extensibility (in the o-o sense), and try to take it to extremes. **Extensible records**.

Running example: Points

- First, define **points** of coords **x,y**, with **m**(-ove) and **eq**(-ual) methods. (Let's do it in Modula-3.)

```
TYPE Point =
  OBJECT
    x,y: INTEGER;
  METHODS
    m(dx,dy: INTEGER): Point;
    eq(other: Point): BOOLEAN;
  END;

PROCEDURE MovePoint(self: Point;  dx,dy:INTEGER): Point =
  BEGIN
    self.x := self.x+dx;  self.y:=self.y+dy;
    RETURN self;
  END MovePoint;

PROCEDURE EqPoint(self,other: Point): BOOLEAN =
  BEGIN
    RETURN (self.x=other.x) AND (self.y=other.y);
  END EqPoint;

VAR p: Point :=
  NEW(Point, x:=0, y:=0, m:=MovePoint, eq:=EqPoint);
```


- Then, define **color points** as points with an additional component: **c**(-olor).

```
TYPE ClrPoint =
  Point OBJECT
    c: Clr;
  END;

VAR cp: ClrPoint :=
  NEW(ClrPoint, x:=0, y:=0, c:=Clr.Black,
    m:=MovePoint, eq:=EqPoint);
```

✌ G o-o d news:

- We reuse and extend the definition of *Point*.
- We have subtyping: every *ClrPoint* is a *Point*.
- We inherit the code of *MovePoint* and *EqPoint*.

☛ Bad news: later.

The main o-o typing trick

Suppose we have:

cp: ClrPoint

where the *m* method of *cp* changes also the color *c*. By subclassing:

cp: Point

Now, if we could extract the **raw procedure** *cp*⇝*m* which was provided as the method *m* of a point, we would have:

$cp \rightsquigarrow m : Point \to Int \times Int \to Point$
$cp \rightsquigarrow m(p)(n,m)$

CRASH! Whenever the point *p* lacks *c*.

Fortunately, o-o languages (starting with Simula) forbid the extraction of raw procedures. Subclassing remains sound because of the following invariant:

The self parameter of a method is always the object from which the method is extracted.

$o.m \equiv o \rightsquigarrow m(o)$

Now, this is an invariant about object **values**, which leaves us with a fundamentally diffucult choice when trying to reduce o-o **typing** to "something simpler":

- Either we have object types built in at the lowest level of the formalism (as in o-o languages), so that the invariant is maintained via rules about object types.
- Or we build objects types from more primitive concepts, and we must find some other way to enforce the invariant, or something equivalent.

The latter is extremely difficult. Nonetheless, this is the road we shall follow.

Type systems fundamentals

- 1st-order types (**System F_1**)
  (data structures and higher-order functions)

  $Nat, A \times B, A+B, A \to B; \quad \mu(X)B$

  (Adding subtyping:
  watch out for $\to$ and μ)

- 2nd-order types (**System F_2** or ***F***)
  (ML polymorphism, CLU a.d.t.'s, and more)

  $X, A \to B, \forall(X)B; \quad \mu(X)B$

  (Nat, $\times$, $+$, $\exists$ are definable)

  (Adding subtyping:
  bounded quantification: $\forall(X<:A)B$
  F-bounded quantification: $\forall(X<:F[X])B$
  meet types: $A \wedge B <: A$)

1st-order record types

Programming with $\times$ and + is extremely boring. In practice we want to use **labeled**, not positional, data structures. These arise frequently in languages as enumerations, records, modules, ... and objects.

- Generalize products $A_1 \times .. \times A_n$ to unordered labeled tuples $(l_1{:}A_1, .., l_n{:}A_n)$.
- Subtype enrichment: require $(l_1{:}A, l_2{:}B, l_3{:}C)$ to be considered **as good as** $(l_1{:}A, l_2{:}B)$, $(l_2{:}B)$, etc.

We call the resulting structures **records**, written:

$Rcd(l_1{:}A_1, .., l_n{:}A_n)$	record types, l_i distinct
$rcd(l_1{=}a_1, .., l_n{=}a_n)$	records values, l_i distinct

enjoying a subtyping (<:) property, e.g.:

$$Rcd(l_1{:}A, l_2{:}B, l_3{:}C) <: Rcd(l_1{:}A, l_3{:}C)$$

Note: a similar path may be followed to generalize +, obtaining **variants**.

$$vnt(l_1{=}a_1) : Vnt(l_1{:}A, l_2{:}B, l_3{:}C)$$
$$Vnt(l_1{:}A, l_3{:}C) <: Vnt(l_1{:}A, l_2{:}B, l_3{:}C)$$

(This will not be discussed further.)

Expected properties of subtyping

- Subtyping is a **reflexive** and **transitive** relation. (A preorder; often a partial order, but this is not useful in typechecking.):

$$A<:A, \quad A<:B \wedge B<:C \Rightarrow A<:C$$

- Satisfies **subsumption**; the single rule connecting subtyping assertions with typing assertions:

$$a:A \wedge A<:B \Rightarrow a:B$$

- Is **structural** over type constructors; the subtyping of the whole depends only on the subtyping of the parts.

$A<:A' \wedge B<:B' \Rightarrow A{\times}B <: A'{\times}B'$	**hierarchical**
$A'<:A \wedge B<:B' \Rightarrow A{\rightarrow}B <: A'{\rightarrow}B'$	**contravariant**
$(X<:Y \Rightarrow A<:B) \Rightarrow \mu(X)A <: \mu(Y)B$	**infinite-unfold**

Ex: Points (via 1st-order records)

Let Point =
$\mu(Self)\ Rcd(x,y{:}Int,\ m{:}Int{\times}Int{\rightarrow}Self)$

Let ClrPoint =
$\mu(Self)\ Rcd(x,y{:}Int,\ c{:}Clr,\ m{:}Int{\times}Int{\rightarrow}Self)$

$Point \equiv Rcd(x,y{:}Int,\ m{:}Int{\times}Int{\rightarrow}Point)$
$ClrPoint \equiv Rcd(x,y{:}Int,\ c{:}Clr,\ m{:}Int{\times}Int{\rightarrow}ClrPoint)$

✌ Good news: *ClrPoint <: Point*

?! Weird: the above fails if we include *eq* methods.

☛ Bad news: *ClrPoint* does not reuse *Point*.
Even if we say *Let ClrPoint = Point ∥ Rcd(c:Clr)*,
the result type of *m* is unsatisfactory for *ClrPoint*.

A design niche

We have reached a clear-cut point in design space: a 1st-order language featuring records and subtyping:

$$Nat,\ Rcd(l_1{:}A_1, .., l_n{:}A_n),\ Vnt(l_1{:}A_1, .., l_n{:}A_n),$$
$$A{\rightarrow}B,\ \mu(X)B$$

The next natural step is to add polymorphism. But this is not all that easy.

2^{nd}-order record types

- Prologue: **2^{nd}-order types** are types parameterized by **type variables**:

    $length: \forall(X)List(X) \to Nat$

Type variables can be instantiated with types, e.g. *Nat*:

$length(Nat): List(Nat) \to Nat$

$length(Nat)([1,2,3]) = 3$
$length(Bool)([true,false]) = 2$

- **2^{nd}-order record types** (perhaps a slight misnomer) are record types parameterized by **type-row** (or **row**, or **extension**) **variables**.

    $Rcd(l_1{:}A_1, .., l_n{:}A_n, X)$

where X is a type-row variable that can be instantiated with an appropriate **type-row**, e.g. $l_{n+1}{:}A_{n+1}, Y$:

$Rcd(l_1{:}A_1, .., l_n{:}A_n, l_{n+1}{:}A_{n+1}, Y)$

The **empty** (or more appropriately, **uninteresting**) type-row is called *Etc*. We can use it to finally instantiate Y above:

$Rcd(l_1{:}A_1, .., l_n{:}A_n, l_{n+1}{:}A_{n+1}, Etc)$

Ex: Points (via 2^{nd}-order records)

Let Point[X ...] = (see later about the "...")
$\mu(Self)\ Rcd(x,y{:}Int, m{:}Int \times Int \to Self, X)$

$Point[X] \equiv Rcd(x,y{:}Int, m{:}Int \times Int \to Point[X], X)$

Let ClrPoint[Y ...] = Point[c:Clr, Y]

ClrPoint[Y]
$\equiv \mu(Self)\ Rcd(x,y{:}Int, c{:}Clr, m{:}Int \times Int \to Self, Y)$
$\equiv Rcd(x,y{:}Int, c{:}Clr, m{:}Int \times Int \to ClrPoint[Y], Y)$

✌ Good news:

- *ClrPoint[Etc] <: Point[Etc]*
- *ClrPoint* reuses the definition of *Point*
- *m* is parametric over extensions of *Point*

Part 2. Back to foundations.

A more detailed understanding of the modeling features we seem to need.

How to reduce them to more basic notions.

Note: this part is non-standard. Different foundational approaches are used in the literature.

Type rows

In general, a 2nd-order record type has the form:

Rcd(R)

where *R* is a type-row; that is, either:

X	type-row variable
Etc	uninteresting type-row
l:A, R	type-row with *l:A*, followed by *R*

But what happened to the restriction that labels in a record must be distinct?

- First, *l:A, R* can be well-formed only if *l* does not occur in *R*. This is written $R\upharpoonleft l$:

$R\upharpoonleft L$ *R* **lacks (exactly)** $L \equiv l_1, .., l_n, n\geq 0.$

- The notion of "lacks" must be respected under substitution, so *l:A, X* requires:

$X\upharpoonleft l$ i.e. *X* can be instantiated only to type-rows *R* such that $R\upharpoonleft l$.

- The idea of "lacks" must be applicable to the *Etc* type-row. Consider:

l:A, Etc	requires $Etc\upharpoonleft l$
$l_1{:}A_1, l_2{:}A_2, Etc$	requires $Etc\upharpoonleft l_1, l_2$

Hence, we need to assume that *Etc* lacks (exactly) anything we want, or perhaps that there are multiple versions of *Etc* indexed by what they lack.

- Only a **complete** row can give raise to a record:

Rcd(R) requires $R\upharpoonleft ()$ (*R* lacks nothing)

"complete" or "lacks nothing" does not mean every label is defined; it means every label is accounted for, either as a field or in the *Etc* sink.

- Finally, wherever there is a type variable there should be a corresponding quantifier. So:

$\forall(Y\upharpoonleft l)B$ for all type-rows *Y* lacking *l* ...

Exercise: if you think this is strange, there are alternative approaches. Try and formalize a similar notion of **lacks at least** or separate notions of **has** and **lacks**.

Value rows

At the value level, we have a notion of (**value-**) **rows** for record values:

rcd(r)

where *r* is a row; that is, either:

x	row variable
etc	uninteresting row
l=a, r	row with *l=a*, followed by *r*
a\L	row of record *a* minus all *L* fields

where now:

$r \therefore R\upharpoonleft L$ means *r* **has** *R* and **lacks (exactly)** *L*

- Wherever there is a value variable there should be a corresponding function space. So:

$R\upharpoonleft L \rightarrow B$ functions from rows to values

Technical examples

- $etc \;\therefore\; Etc{\uparrow}l$ an axiom

  $l{=}3,etc \;\therefore\; l{:}Nat,Etc{\uparrow}()$

  $rcd(l{=}3,etc) \;:\; Rcd(l{:}Nat,Etc)$

  $rcd(l{=}3,etc).l \;:\; Nat$

- $x \;:\; Rcd(l{:}Nat,Y)$ an assumption ($\Rightarrow Y{\uparrow}l$)

  $x\backslash l \;\therefore\; Y{\uparrow}l$

  $l{=}x.l{+}1,\, x\backslash l \;\therefore\; l{:}Nat,Y{\uparrow}()$

  $rcd(l{=}x.l{+}1,\, x\backslash l) \;:\; Rcd(l{:}Nat,Y)$

  $\lambda(x{:}Rcd(l{:}Nat,Y))\; rcd(l{=}x.l{+}1,\, x\backslash l)$
  $\;:\; Rcd(l{:}Nat,Y) \to Rcd(l{:}Nat,Y)$

  $\lambda(Y{\uparrow}l)\; \lambda(x{:}Rcd(l{:}Nat,Y))\; rcd(l{=}x.l{+}1,\, x\backslash l)$
  $\;:\; \forall(Y{\uparrow}l)\; Rcd(l{:}Nat,Y) \to Rcd(l{:}Nat,Y)$

Is this the right calculus?

When fully formalized, the calculus with extensible records described so far is called $F_{<:}\rho$ and has a total of 78 typing and evaluation rules. Rather complicated!

Several other formulations of extensible records have been proposed, and have a comparable number of rules.

Is this the **right** calculus? Not clear. However, what **distinguishes** $F_{<:}\rho$ is that it can be completely encoded into a much simpler calculus called $F_{<:}$ which has "only" 32 rules ($F_{<:}\rho$ is in fact an extension of $F_{<:}$). We remain **within pure 2nd-order** calculi.

By a comparable way of counting: F_2 ($\equiv F$, the polymorphic or 2nd-order λ-calculus) has 22 rules; F_1 (the simply-typed or 1st-order λ-calculus) has 14 rules; and the untyped λ-calculus (F_0 ?) has 10 rules.

A pure calculus of subtyping: $F_{<:}$

$F_{<:}$ is obtained by starting with 2nd-order types and adding subtyping, with *Top* the biggest type.

$$X,\ Top,\ A{\to}B,\ \forall(X{<:}A)B;\quad \mu(X)B$$

For terms of the calculus we have

$$x,\ top,\ \lambda(x{:}A)b,\ b(a),$$
$$\lambda(X{<:}A)b,\ b(A);\quad \mu(x{:}A)b$$

Unlike *F*, **equivalence** of two terms in $F_{<:}$ is stated always **with respect to a type**. The type acts as an observer. Values that are distinguishable in a subtype may become undistinguishable in a supertype (this is characteristic of objects). At the limit, everything is undistinguishable in *Top*.

Models: partial equivalence relations (per's) over $(\omega,\cdot)$, where $<:$ is $\subseteq$ of per's. For recursion: per's over D_∞.

Soundness of $F_{<:}\rho$

Theorem There is a translation of $F_{<:}\rho$ into $F_{<:}$ that preserves all derivations (typing, subtyping, and equivalence).

Hint.

- Using a standard technique from *F* we can encode cartesian products $A{\times}B$ in $F_{<:}$ (which are automatically monotonic w.r.t. $<:$).
- From these, we can define:

$$Tuple(A_1,..,A_n,B) = A_1{\times}..{\times}A_n{\times}B$$

Consider tuples where the final $B{\equiv}Top$; then a "longer" tuple is a subtype of a "shorter" tuple.

- Fix an enumeration of labels. Translate records to tuples according to the index of labels, e.g.:

$$Rcd(l^2{:}C, l^0{:}A, Top) \equiv Tuple(A, Top, C, Top)$$

position: 0 1 2 3+

$$Rcd(l^2{:}C, l^0{:}A, X) \equiv Tuple(A, X^1, C, X^3)$$

position: 0 1 2 3+

Under this translation, for records ending with *Top (≡Etc)*, "longer" ones are subtypes of "shorter" ones. Moreover, the order of fields is normalized.

- Finally, type-row variables become rows of type variables; if *X* lacks (exactly!) l^0, l^2, then it has (exactly) l^1 and l^3, l^4.... The tail can be captured by a single variable:

$$\forall(X \Uparrow l^0, l^2)\ldots \equiv \forall(X^1)\forall(X^3)\ldots$$

Following this pattern, type-row applications become rows of type applications, etc.

Part 3. Forward to objects.

Using subtyping and parameterization to (attempt to) emulate o-o constructs.

What's the connection to o-o?

We try to *model* (as well as we can) basic o-o concepts, *explain* them (via "more fundamental notions") and *extend* them (by combining fundamental notions synergicly).

We feel the need for this exercise because not all is well-understood or clear-cut with o-o languages.

We could use a better understanding of o-o concepts for designing new, simpler, and more powerful languages, and to avoid pitfalls (e.g. unsound type systems).

Or maybe, once we truly understand these concepts, we may decide they are too complicated and scrap them...

Remember the Modula-3 code?

☛ Bad news:

- *cp.move* has return type *Point*, not *ClrPoint*, although it really returns a *ClrPoint*. (Note that *MovePoint* could allocated and return a new *Point*, which would certainly not be a *ClrPoint*.)

- Although it would be highly desirable, we **cannot** override *eq* using:

PROCEDURE EqClrPoint(self,other: ClrPoint ...

because Modula-3 requires *other:Point*.

This is a deep problem, not exclusive to Modula-3.

To fix these shortcomings, a language like **Eiffel** might use something like this:

METHODS
m(dx,dy: INTEGER):Self; **covariant *Self***
eq(other: Self): BOOLEAN; **contravariant *Self***

But one has to be ***very careful***: covariant *Self* gives subclasses that are subtypes, but contravariant *Self* gives subclasses that are **not** subtypes (or else the typechecker is unsound). More about this later.

Let's now see how one might paraphrase the *Point* example using extensible records, along with recursion.

Recursion and extension

An **object type** is some kind of recursive record type.

$Let\ A = \mu(S)\ Rcd(n{:}Int,\ f{:}S\to S)$
$Let\ B = \mu(S)\ Rcd(n{:}Int,\ f{:}S\to S,\ g{:}S\to S)$

General problem: how can we define *B* by reusing *A*?

Record concatenation (whatever that means) does not help much:

$Let\ B' = A \parallel \mu(S)\ Rcd(g{:}S\to S)$

here *B′* does not "loop the same way" as *B*.

A solution is to use **generators**, that is to leave the recursion open so we can close it later in the desired way.

$Let\ GenA[S] = Rcd(n{:}Int,\ f{:}S\to S)$
$Let\ A = \mu(S)\ GenA[S]$ (i.e. *Fix(GenA)*)

$Let\ GenB[S] = GenA[S]\ with\ (g{:}S\to S)$
$Let\ B = \mu(S)\ GenB[S]$

(Note: *with* needs to be defined appropriately.)

This technique can be carried quite far. Contravariant Self (e.g. *eq* methods) leads to **F-bounded quantification**.

We do not discuss this further because...

Instead of generators and F-bounded quantification, we can use **record extension** and **parametric definitions**.

We close recursions immediately, but we still manage to patch them later via extensions.

$Let\ ExtA[X\uparrow f] = \mu(SA)\ Rcd(n{:}Int,\ f{:}SA\to SA,\ X)$
$Let\ A = ExtA[Etc]$

$Let\ ExtB[Y\uparrow f,g] = \mu(SB)\ ExtA[g{:}SB\to SB,\ Y]$
$Let\ B = ExtB[Etc]$ *(</: A)*

We have:

$ExtB[Y]$
$\equiv \mu(SB)\ \mu(SA)\ Rcd(f{:}SA\to SA,\ g{:}SB\to SB,\ Y)$
$\equiv \mu(S)\ Rcd(f{:}S\to S,\ g{:}S\to S,\ Y)$

A similar trick works with value-level recursion.

Exercise: do the examples in section 3 of [Cook Hill Canning 90] using only extensible records and parametric definitions.

Ex: Points (via 2nd-order records)

- Points:

Let Point[X↑x,y,m] =
μ(Self) Rcd(x,y:Int, m:Int×Int→Self, X)

let newPoint(W↑x,y,m)(x,y:Int, m:Point[W]→Int×Int→Point[W])
(w∴W): Point[W] =
μ(self:Point[W]) rcd(x=x, y=y, m=m(self), w)

let rec movePoint(W↑x,y,m)(self:Point[W])(dx,dy:Int): Point[W] =
newPoint (W) (self.x+dx, self.y+dy, movePoint(W)) (self \x,y,m)

let p: Point[Etc] =
newPoint (Etc) (0, 0, movePoint(Etc)) (etc)

- Color points inheriting *m* from *Point*.

Let ClrPoint[Y↑x,y,c,m] = Point[c:Clr, Y]
≡μ(Self) Rcd(x,y:Int, c:Clr, m:Int×Int→Self, Y)

let newClrPoint(Z↑x,y,c,m)(x,y:Int, c:Clr,
m:ClrPoint[Z]→Int×Int→ClrPoint[Z])(z∴Z): ClrPoint[Z] =
newPoint(c:Clr,Z)(x, y, m)(c=c, z)

let cp:ClrPoint[Etc] =
newClrPoint(Etc) (0 ,0, black, movePoint(c:Clr,Etc)) (etc)

- Color points overriding *m* from *Point*.

let rec moveClrPoint(Z↑x,y,c,m)(self:ClrPoint[Z])(dx,dy:Int)
: ClrPoint[Z] =
newClrPoint (W) (self.x+dx, self.y+dy, red, moveClrPoint(W))
(self \x,y,c,m)

let cp:ClrPoint[Etc] =
newClrPoint (Etc) (0, 0, black, moveClrPoint(Etc)) (etc)

✌ Good news:

- *movePoint* has a *self* first argument, but this does not show in the type of *Point* because the **procedure** *movePoint* is converted to the **method** *m*.
Otherwise *ClrPoint[Etc] <: Point[Etc]* would fail because of contravariance.

- The *new* routine for *ClrPoint* uses the *new* routine for *Point*. This kind of behavior is useful or necessary to establish the internal invariant of superclasses on allocation of subclasses (e.g., polar points).

- *cp.move* is inherited from *Point*, but has the appropriate return type when used from *ClrPoint*.

?! Weird: *eq* methods don't subtype...

Inheritance without subtyping

If we include the *eq* method in the definitions, obtaining *EqPoint* (and hence *ClrEqPoint*), we can still inherit methods, but then we do **not** have

ClrEqPoint[Etc] <: EqPoint[Etc]

Let's ignore *m*. (See Appendix for the full example.)

Let EqPoint[X↑x,y,eq] =
μ(Self) Rcd(x,y:Int, eq:Self→Bool, X)

Let ClrEqPoint[Y↑x,y,c,eq] = EqPoint[c:Clr, Y]
≡μ(Self) Rcd(x,y:Int, c:Clr, eq:Self→Bool, Y)

The type rules for recursion fail to prove *ClrEqPoint[Etc] <: EqPoint[Etc]* because of the contravariant occurrence of *Self* in *eq*.

Are the type rules too weak? Or is this inclusion really bogus?

Let's assume *ClrEqPoint[Etc] <: EqPoint[Etc]*, and take: *cp:ClrEqPoint[Etc]*, *p:EqPoint[Etc]*. Let's also assume that *cp.eq* tests the *c* components.

Then *cp:EqPoint[Etc]*, by subsumption. Hence:

cp.eq: EqPoint[Etc]→Bool.

Therefore,

cp.eq(p): Bool is well-typed.

But *cp.eq* will access the *c* component of *p*, which *p* does not have: **CRASH!** The type rules were right after all.

Conclusions

- It is important to unbundle subtyping from inheritance. We can take advantage of subtyping without inheritance, and of inheritance without subtyping.

- A language with subtyping and sufficient parameterization (several choices here) can emulate basic o-o concepts and go beyond them. Many of the additional features are natural o-o desiderata.

- On the other hand, it is very difficult to provide in a much simpler way *exactly* what o-o already provides.

Further reading

Highly recommended:

[Cook Hill Canning 90] *Inheritance is Not Subtyping*. Proc. POPL'90.
(Introduction to generators and F-bounded quantification.)

[Bruce 92] *A Paradigmatic Object-oriented Programming Language: Design, Static Typing, and Semantics*. To appear.
(A direct formalization of an interesting o-o language and its typing.)

Appendix. The full example

- Points.

Let Point[X↑x,y,m] =
μ(Self) Rcd(x,y:Int, m:Int×Int→Self, X)

let newPoint(W↑x,y,m)(x,y:Int, m:Point[W]→Int×Int→Point[W])
(w∴W): Point[W] =
μ(self:Point[W]) rcd(x=x, y=y, m=m(self), w)

let rec movePoint(W↑x,y,m)(self:Point[W])(dx,dy:Int): Point[W] =
newPoint (W) (self.x+dx, self.y+dy, movePoint(W)) (self \x,y,m)

let p: Point[Etc] =
newPoint (Etc) (0, 0, movePoint(Etc)) (etc)

- Color points inheriting *m* from *Point*.

Let ClrPoint[Y↑x,y,c,m] = Point[c:Clr, Y]
≡μ(Self) Rcd(x,y:Int, c:Clr, m:Int×Int→Self, Y)

let newClrPoint(Z↑x,y,c,m)(x,y:Int, c:Clr,
m:ClrPoint[Z]→Int×Int→ClrPoint[Z])(z∴Z): ClrPoint[Z] =
newPoint(c:Clr,Z)(x, y, m)(c=c, z)

let cp:ClrPoint[Etc] =
newClrPoint(Etc) (0 ,0, black, movePoint(c:Clr,Etc)) (etc)

- Points with *eq*, reusing *m* from *Point*.

```
Let EqPoint[X↑x,y,m,eq] =
  μ(Self) Point[eq:Self→Bool, X]
  ≡ μ(Self) Rcd(x,y:Int, m:Int×Int→Self, eq:Self→Bool, X)

let newEqPoint(W↑x,y,m,eq) (x,y:Int,
      m:EqPoint[W]→Int×Int→EqPoint[W],
      eq:EqPoint[W]→EqPoint[W]→Bool) (w∴W): Point[W] =
  μ(self:EqPoint[W])
      newPoint(eq:EqPoint[W]→Bool,W)(x,y,m)(eq=eq(self),w)

let rec eqEqPoint(W↑x,y,m,eq)(self:EqPoint[W])(other:EqPoint[W])
      : Bool =
  self.x=other.x & self.y=other.y

(* A movePoint "wrapper", so that p.m(2,3).eq(p)=false *)
let rec moveEqPoint(W↑x,y,m,eq)(self:EqPoint[W])(dx,dy:Int)
      : EqPoint[W] =
  let p = movePoint(eq:EqPoint[W]→EqPoint[W]→Bool,W)(self)(dx,dy)
  in μ(self':EqPoint[W]) rcd(eq=eqEqPoint(W)(self'), p\eq)

let ep: EqPoint[Etc] =
  newEqPoint(Etc)(0, 0, moveEqPoint(Etc), eqEqPoint(Etc)) (etc)
```


- Points with *eq* and *c*, inheriting *m* from *EqPoint*, and overriding (but still reusing) *eq* from *EqPoint*. But **not** *ClrEqPoint[Etc] <: EqPoint[Etc]*.

```
Let ClrEqPoint[Y↑x,y,c,m,eq] =
  EqPoint[c:Clr, Y]
  ≡μ(Self) Rcd(x,y:Int, c:Clr, m:Int×Int→Self, eq:Self→Bool, Y)

let newClrEqPoint(Z↑x,y,c,m,eq)
    (x,y:Int, c:Clr,
     m:ClrEqPoint[W]→Int×Int→ClrEqPoint[W],
     eq:ClrEqPoint[Z]→ClrEqPoint[Z]→Bool)
    (z∴Z): ClrEqPoint[Z] =
  newEqPoint(c:Clr,Z)(x, y, m, eq)(c=c,z)

let rec eqClrEqPoint(W↑x,y,m,c,eq)(self:ClrEqPoint[W])
    (other:ClrEqPoint[W]): Bool =
  eqEqPoint(c:Clr,Etc)(self)(other) & self.c=other.c

let cep:ClrEqPoint[Etc] =
  newClrEqPoint(Etc)
    (0, 0, black, moveEqPoint(c:Clr, Etc), eqClrEqPoint(Etc)) (etc)
```


Advert. $F_{<:}$ software

Fsub is a Modula-3 implementation of the $F_{<:}$ calculus. This is the "smallest possible" calculus integrating subtyping with polymorphism. The type structure consists of type variables, "Top", function spaces, bounded quantification, and recursive types. The implementation supports type inference ("argument synthesis"), a simple modularization mechanism, and the introduction of arbitrary notation on the fly.

The system can be obtained by anonymous ftp from **gatekeeper.pa.dec.com**, in the **DEC** directory. The distribution includes DECstation and VAX binaries; it can be ported to other architectures that support Modula-3 by recompilation.

The Fsub licence is covered by the Modula-3 licence; there is nothing to sign. If needed, Modula-3 can be obtained by anonymous ftp from gatekeeper.pa.dec.com.

A manual "**F-sub, the system**" is included in postscript format. Hardcopies may be obtained from:

Luca Cardelli (luca@src.dec.com)
DEC SRC, 130 Lytton Ave
Palo Alto, CA 94310, USA

References on selected topics

First-order subtyping and simple records.

[Mitchell 84] J.C.Mitchell: **Coercion and type inference**, Proc. of the 11th ACM Symposium on Principles of Programming Languages, pp.175-185, 1984.

[Cardelli 84] L.Cardelli: **A semantics of multiple inheritance**, in Semantics of Data Types, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[Reynolds 88] J.C.Reynolds: **Preliminary design of the programming language Forsythe**, Report CMU-CS-88-159, Carnegie Mellon University, 1988.

Recursive type equivalence and subtyping.

[Breazu-Tannen et al. 89] V. Breazu-Tannen, C. Gunter, A. Scedrov: **Denotational semantics for subtyping between recursive types**, Report MS-CIS 89 63, Logic of Computation 12, Dept of Computer & Information Science, University of Pennsylvania.

[Amadio Cardelli 91] R.M.Amadio, L.Cardelli: **Subtyping recursive types**, Proceedings of the ACM conference on Principles of Programming Languages, ACM Press, 1991.

Second-order typing.

[Girard 71] J-Y.Girard: **Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types**, Proceedings of the second Scandinavian logic symposium, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Reynolds 74] J.C.Reynolds: **Towards a theory of type structure**, in Colloquium sur la programmation pp. 408-423, Springer-Verlag Lecture Notes in Computer Science, n.19, 1974.

[Mitchell Plotkin 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, Proc. POPL 1985.
[Scedrov 90] A.Scedrov: **A guide to polymorphic types**, in Logic and Computer Science, pp 387-420, P.Odifreddi ed., Academic Press, 1990.

Second-order subtyping and simple records.
[Cardelli Wegner 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.
[Breazu-Tannen Coquand Gunter Scedrov 89] V.Breazu-Tannen, T.Coquand, C.Gunter, A.Scedrov: **Inheritance and explicit coercion** , Proc. of the Fourth IEEE Symposium on Logic in Computer Science, pp 112-129, 1989.

Pure second-order subtyping.
[Ghelli 90] G.Ghelli: **Proof theoretic studies about a mininal type system integrating inclusion and parametric polymorphism**, Ph.D. Thesis TD-6/90, Università di Pisa, Dipartimento di Informatica, 1990.
[Curien Ghelli 91] P.-L.Curien, G.Ghelli: **Coherence of subsumption**, Mathematical Structures in Computer Science, to appear.
[Curien Ghelli 91] P.-L.Curien, G.Ghelli: **Subtyping + extensionality: confluence of $\beta\eta$-reductions in $F_{\leq}$**, in T.Ito,A.R.Meyer Eds.Theoretical Aspects of Computer Software, Sendai, Japan, Lecture Notes in Computer Science n.526, pp. 731-749, Springer-Verlag, 1991.
[Cardelli Martini Mitchell Scedrov 91] L.Cardelli, J.C.Mitchell, S.Martini, A.Scedrov: **An extension of system F with subtyping**, in T.Ito,A.R.Meyer Eds.Theoretical Aspects of Computer Software, Sendai, Japan, Lecture Notes in Computer Science n.526, pp. 750-770, Springer-Verlag, 1991.
[Pierce 91] B.C.Pierce: **Programming with intersection types and bounded polymorphism**, Ph.D. Thesis, CMU-CS-91-205, 1991.
[Pierce 92] B.C.Pierce: **Bounded quantification is undecidable**, Proc. POPL'92

Generators and F-bounded quantification.
[Reddy 88] U.S.Reddy: **Objects as closures: abstract semantics of object-oriented languages**, Proc. ACM Conference on Lisp and Functional Programming, pp. 289-297, 1988.
[Cook 89] W. Cook: **A denotational semantics of inheritance**, Ph.D. thesis, Technical Report CS-89-33, Brown University, 1989.
[Canning Hill Olthoff 88] P.Canning,W.Hill, W.Olthoff: **A kernel language for object-oriented programming**, Technical Report STL-88-21, Hewlett-Packard Labs, 1988.
[Canning Cook Hill Olthoff Mitchell 89] P.Canning, W.Cook, W.Hill, W.Olthoff, J.C.Mitchell: **F-bounded polymorphism for object-oriented programming**, Proc. ACM Conference on Functional Programming and Computer Architecture, ACM Press, 1989.
[Cook Hill Canning 90] W.Cook, W.Hill, P.Canning: **Inheritance is not subtyping**, Proc. POPL'90.

Extensible records.
[Wand 87] M.Wand: **Complete Type Inference for Simple Objects**, Proc. of the Second IEEE Symposium on Logic in Computer Science, pp 37-44, 1987. **Corrigendum: Complete Type Inference for Simple Objects**, Proc. of the Third IEEE Symposium on Logic in Computer Science, 1988.
[Jategaonkar Mitchell 88] L.A.Jategaonkar, J.C.Mitchell: **ML with extended pattern matching and subtypes**, Proc. of the ACM Conference on Lisp and Functional Programming, pp.198-211, 1988.
[Wand 89] M.Wand: **Type inference for record concatenation and multiple inheritance**, Proc. of the Fourth IEEE Symposium on Logic in Computer Science, pp. 92-97, 1989.
[Rémy 89] D. Rémy: **Typechecking records and variants in a natural extension of ML**, Proc. of the 16th ACM Symposium on Principles of Programming Languages, pp.77-88, 1989.

[Cardelli Mitchell 91] L.Cardelli, J.C.Mitchell: **Operations on records**, Mathematical Structures in Computer Science, vol 1, pp.3-48, 1991.
[Harper Pierce 90] R.Harper, B.Pierce: **A record calculus with symmetric concatenation**, Technical Report CMU-CS-90-157, CMU, 1990.
[Cardelli 91] L.Cardelli: **Extensible records in a pure calculus of subtyping**, DEC SRC Report #81, 1991.
[Rémy 92] D. Rémy: **Typing record concatenation for free**, Proc. of the 19th ACM Symposium on Principles of Programming Languages, 1992.

Type inference for subtyping and o-o languages.
Many references.
Semantics of o-o langauges.
Many references.
Models of subtyping.
Many references.