INTRODUCTION

Object-oriented languages have a large number of built-in concepts: Classes, Subclasses, Methods, Self, Message-Passing, Inheritance, Delegation, Overriding, etc.

Can we understand these concepts in terms of (a smaller number of) more basic concepts?

Can we turn this new understanding into better and more general programming features?

We try this in the context of (extension of) second-order lambda-calculus.

(Second-order lambda-calculus has been successfully used to model explicitly-polymorphic languages, i.e. languages where one can explicitly abstract over types).

OUTLINE

PART 1
- Records
- Record types
- Record kinds
- Record value variables
- Record type variables
- Making a disc
- Moving a point
- Computing lumens
- Record subtyping

PART 2
- Subsumption
- Loss of genericity
- Bounded Quantification

PART 3
- Contravariance
- Non-generic Methods
- Method: Lumens of a point
- Method: Moving a point
- Non-generic Classes

PART 4
- Generic Methods
- Generic Classes

CONCLUSIONS

APPENDIX: Type Rules


**PART I**

**RECORD TYPES AND SUBTYPES**

**Records**

- Empty record
  - `{}`
  - `{a=3, b=true}`
- Record with two fields
  - `{b=true, a=3}`
  - `{a=3, b=true}`
  - `{a=3, b=true}\b`
  - `{a=3, b=true}\c
  - `{a=3} <- a=true}
  - `{a=3, b=true}.a
- Record with three fields
  - `{a=3, a=4}
  - `{a=3} | a=4}
  - `{a=3} .b

**Record types**

- Records with at least some fields
  - `{a=3, b=true, c="ab"}, {a=3}`
- Records without some fields
  - `{a=3, c="ab"}, {a=3, b=true}`
- All records
  - `{}, {a=3}, etc.`
- `{b:Boolean, a:Integer}`
- `{a:Integer, b:Boolean}|b`
- `{a:Integer} | b:Boolean`
- `{a:Integer} <- a:Boolean`
- `{a:Integer, a:Boolean}
- `{a:Integer} | a:Boolean}
- `{a:Integer} | b:Boolean`

**Record kinds**

- `::TYPE the kind of all types`
- `::[] the kind of all record types (e.g. `{a:b:Integer\c, Integer})`
- `::[a,b] the kind of all record types that have some components (e.g. `{a:b:Integer\c, {a:Integer})`
- `::[]\a\b the kind of those record types that do not have a or b components (e.g. `{c:Integer\a \b)"
Record value variables

```plaintext
{{{{r | b=true}}}}  r must not have b (i.e. r: \(\{\}\)\b)
```r . a  r must have a (i.e. r: \(\{a:T\}\))
\r/c  r may or may not have c

The information about what r must or must not have is "remembered" in its type:

```plaintext
let f(r: {{{{{a:Int}}}}}\b): {{{{{a:Int, b:Int}}}}} =
{{{{r <- a=r.a+1 | b=0}}}}
f : {{{{{a:Int}}}}}\b -> {{{{{a:Int, b:Int}}}}}
```

Record type variables

```plaintext
:{{{{{R| c:Bool}}}}}R  R must not have c
```let f(r: {{{{{R| a:Int}}}}}): {{{{{R| a b:Int}}}}} =
{{{{r <- a=r.a+1 | b=0}}}}
f : {{{{{R| a:Int}}}}} -> {{{{{R| a b:Int}}}}}

But where is R (which must not have a or b) bound? The information about what R must and must not have is "remembered" in its kind:

```plaintext
let f(R::{{{{{}}}}}\a b)(r: {{{{{R| a:Int}}}}}): {{{{{R| a b:Int}}}}} =
{{{{r <- a=r.a+1 | b=0}}}}
f : Fun(R::{{{{{}}}}}\a b) {{{{{R| a:Int}}}}} -> {{{{{R| a b:Int}}}}}
```

Making a disc

This is an example of a situation where one wants to embed a structure into a bigger one. On subtypes, the domain and range types change in unison, and the semantics stays the "same".

Let Point = {{{{{x y: Int}}}}}
Let ColorPoint = {{{{{Point <- rgb: RGB}}}}}
Let Disc = {{{{{Point <- r: Int}}}}}
Let ColorDisc = {{{{{ColorPoint <- r: Int}}}}}

```plaintext
let p: Point = {{{{{x=3, y=4}}}}}
let cp: ColorPoint = {{{{{p <- rgb=red}}}}}
```

Let discOfPoint(p: {{{{Point\r}}}}): Disc =
{{{{x=p.x, y=p.y, r=1}}}} or {{{{p| r=1}}}}

let cDiscOfCPoint(cp: {{{{ColorPoint\r}}}}): ColorDisc =
{{{{cp| r=1}}}}
Moving a point

This is an example of a situation where one wants to change an existing structure. On subtypes, the domain and result types change in unison, and the semantics stays the "same".

Let \( P = \{x, y: \text{Int}\} \)
Let \( CP = \{\text{Point} \leftarrow \text{rgb}: \text{RGB}\} \)

let \( p: P = \{x=3, y=4\} \)
let \( cp: CP = \{p \leftarrow \text{rgb} = \text{red}\} \)

Applicative style:

let incXofPoint\((p: P)\): P =
\( p \leftarrow x=p.x+1 \)

let incXofCP\((cp: CP)\): CP =
\( cp \leftarrow x=cp.x+1 \)

Pattern-matching style:

let incXofPoint\((\{x=px, y=py\}: P)\): P =
\( \{x=px+1, y=py\} \)

let incXofCP\((\{x=px\}: P)\): P =
\( \{x=px+1\} \)

Imperative style:

let incXofPoint\((p: P)\): P =
\( p.x := p.x+1 \)

Computing lumens

This is an example of a function that changes domain type and semantics on subtypes; the result type is constant.

Let \( P = \{x, y: \text{Int}\} \)
Let \( CP = \{\text{Point} \leftarrow \text{rgb}: \text{RGB}\} \)
Let \( D = \{\text{Point} \leftarrow r: \text{Int}\} \)
Let \( CD = \{\text{ColorPoint} \leftarrow r: \text{Int}\} \)

let lumensOfPoint\((p: P)\): \text{Real} =
\( \text{pixelLumens} \)
let lumensOfCP\((cp: CP)\): \text{Real} =
\( \text{lumensOfPoint}(cp) \times \text{rgbIntensity}(cp.rgb) \)
let lumensOfDisc\((d: D)\): \text{Real} =
\( \text{lumensOfPoint}(d) \times \pi \times d.r \times d.r \)
let lumensOfCD\((cd: CD)\): \text{Real} =
either \( \text{lumensOfCP}(cd) \times \pi \times cd.r \times cd.r \)
or \( \text{lumensOfDisc}(cd) \times \text{rgbIntensity}(cd.rgb) \)

Record subtyping

Let \( P = \{x, y: \text{Int}\} \)
Let \( CP = \{\text{Point} \leftarrow \text{rgb}: \text{RGB}\} \)
then: \( \text{ColorPoint} \leftarrow \text{Point} \) subtyping

because \( \text{ColorPoint} \) has all the attributes of \( \text{Point} \). This is subtyping in "width".

So, \( \text{Point} \) is the type of all the records which have at least \( x \) and \( y \) components.

Now, \( \text{Point}\{r\} \) is the type of all the records which have at least \( x \) and \( y \), but no \( r \).

hence: \( \text{Point}\{r\} \leftarrow \text{Point} \)
Structural subtyping:

Whether:
Let Point = \{x y: \text{Int}\}
Let ColorPoint = \{Point \leftarrow \text{rgb: RGB}\}

or:
Let Point = \{x y: \text{Int}\}
Let ColorPoint = \{x y: \text{Int, rgb: RGB}\}

still:  ColorPoint <: Point

because ColorPoint has all the attributes of Point, no matter how they are constructed.

Hierarchical subtyping:

Let Rect = \{tl br: Point\}
Let ColorRect = \{tl: ColorPoint, br: Point\}

ColorRect <: Rect
because all the attributes of ColorRect are subtypes of the respective attributes of Rect. This is subtyping in "depth.”

Multiple subtyping:

Let Disc = \{Point \leftarrow r: \text{Int}\}
Let ColorDisc = \{ColorPoint \leftarrow r: \text{Int}\}

Subsumption

If \( a:A \) and \( A <: B \) then \( a:B \)

Let Point = \{x y: \text{Int}\}
Let ColorPoint = \{Point \leftarrow \text{rgb: RGB}\}
Let Disc = \{Point \leftarrow r: \text{Int}\}
Let ColorDisc = \{ColorPoint \leftarrow r: \text{Int}\}

let p: Point = \{x=0, y=0\}
let cp: ColorPoint = \{p \leftarrow \text{rgb=red}\}
let d: Disc = \{p \leftarrow r=1\}
let cd: ColorDisc = \{cp \leftarrow r=1\}

Then, by subsumption:

\[
\begin{align*}
\text{cp} & \colon \text{Point} \\
\text{d} & \colon \text{Point} \\
\text{cd} & \colon \text{Disc, cd: ColorPoint, cd: Point}
\end{align*}
\]

Also:

\[
\text{cp}\backslash r: \text{Point}\backslash r
\]
Loss of genericity

The crude use of subsumption causes loss of genericity:

Making a disc:

\[
\text{let discOfPoint}(p: \text{Point}) : \text{Disc} = \{p | r=1\}
\]
Then \(\text{discOfPoint}(cp)\) is legal, by subsumption.
Note however that \(\text{discOfPoint}(cp) : \text{Disc}\).

Moving a point:

\[
\text{let incXofPoint}(p: \text{Point}) : \text{Point} = \{p <- x=p.x+1\}
\]
Then \(\text{incXofPoint}(cp)\) is legal, by subsumption.
Note however that \(\text{incXofPoint}(cp) : \text{Point}\).

Bounded Quantification

We can recover genericity by abstracting over types, so that we can express input-output type dependencies.

We can already abstract over all types belonging to a given kind, as we have seen:

\[
\text{let } f(R::\{\} | x, r:R|x:|Int\}) : \{R|x:|Int\} = \ldots
\]
We shall also allow to abstract over all the subtypes of a given type, by bounded quantification:

\[
\text{let } f(R::\text{Point}, r:R) : \{R<-x:|Int\} = \ldots
\]
(These two ways of abstracting over types are actually special cases of a single mechanism, which we won't get into here.)

Making a disk, generically

We want a disc-making function that returns a \text{Disc} when given a \text{Point}, and a \text{ColorDisc} when given a \text{ColorPoint}.

\[
\text{let discfy}(P<:\text{Point}, p:P) : \{P<-r:|Int\} \equiv \{p<- r=1\}
\]
\[
\text{discfy}(\text{Point}, p) : \{\text{Point}<-r:|Int\} \equiv \text{Disk}
\]
\[
\text{discfy}(\text{ColorPoint}, cp) : \{\text{ColorPoint}<-r:|Int\} \equiv \text{ColorDisc}
\]
(Another possibility:
Let \(\text{discfy}(P<:\text{Point}|r, p:P) : \{P | r:|Int\} = \{p | r=1\}
\]
\[
\text{discfy}(\text{Point}|r, p|r)
\]

Moving a point, generically

We want an increment-x function that returns a \text{Point} when given a \text{Point}, a \text{ColorPoint} when given a \text{ColorPoint}, etc.

\[
\text{let incX}(R<:\text{Point}, r:R) : \{R<-x:|Int\} \equiv \{r <- x=r.x+1\}
\]
\[
\text{incX}(\text{Point}, p) : \text{Point}
\]
\[
\text{incX}(\text{ColorPoint}, cp) : \text{ColorPoint}
\]
\[
\text{incX}(\text{Disc}, p) : \text{Disc}
\]
\[
\text{incX}(\text{ColorDisc}, cp) : \text{ColorDisc}
\]
Note: why not

\[
\text{let } \text{incX}(R <: \text{Point}, r: R): R = \\
\{ r' <- x=r.x+1 \}
\]

Take \([0..9] <: \text{Int}\) so that:

\[
\text{PointLT10} = \{x,y: [0..9]\} <: \text{Point}
\]

\text{incX}(\text{PointLT10}, 9) \text{ does not have type } \text{PointLT10}

(It is not easy, although possible, to place run-time check for <10. Anyway, similar examples can be constructed with other types in place of integers and subranges, for which such checks become arbitrarily complex.)

This shows that the restriction and extension operators are necessary, even if we already have bounded quantification.

\[
\textbf{Contravariance}
\]

Every function returning a color point also returns a point (ignoring the color):

\[
f: T \rightarrow \text{ColorPoint} \Rightarrow f: T \rightarrow \text{Point}
\]

Every function accepting a point also accepts a disc (ignoring the radius):

\[
f: \text{Point} \rightarrow U \Rightarrow f: \text{Disc} \rightarrow U
\]

Combining the two:

\[
f: \text{Point} \rightarrow \text{ColorPoint} \Rightarrow f: \text{Disc} \rightarrow \text{Point}
\]

Therefore:

\[
\text{Point} \rightarrow \text{ColorPoint} <: \text{Disc} \rightarrow \text{Point}
\]

\[
\textbf{In general:}
\]

\[
T \rightarrow U \Rightarrow T' \rightarrow U'
\]

\[
\text{Point} \rightarrow \text{ColorPoint} \Rightarrow \text{Disc} \rightarrow \text{Point}
\]

\[
\text{Point} \rightarrow \text{ColorPoint} <: \text{Disc} \rightarrow \text{Point}
\]
Non-generic Methods

General idea: sending a message \( m \) to an object \( r \).

\[
\text{for } r : R, \quad r <= m \equiv r.m(r)
\]

where \( m : R \rightarrow T \) is a method (its first parameter is \( \text{self} \)).

(A common case: \( T = R \).)

Hence \( R = \{ \ldots \text{method } m : R \rightarrow T \ldots \} \).

Now consider a subtype \( S <: R \), then we should have:

\[
\text{for } s : S, \quad s <= m \equiv s.m(s)
\]

where \( m : S \rightarrow T \). Hence \( S = \{ \ldots \text{method } m : S \rightarrow T \ldots \} \).

But \( S \rightarrow T \) is not a subtype of \( R \rightarrow T \) (the opposite is true, because of contravariance) hence it is false that \( S <: R \) (because of their \( m \) component), contradiction!

Hence we extend records with special "method" components, satisfying the following class-form:

\[
R = \{ \ldots \text{method } m : R' \rightarrow T \ldots \} \quad \text{for } R <: R'
\]

\[
S = \{ \ldots \text{method } m : S' \rightarrow U \ldots \} \quad \text{for } S <: S'
\]

An object is a record with methods.

Methods can only be accessed by the "send" notation:

\[
r <= m \quad (\equiv r.m(r))
\]

Methods have a special subtyping rule, ignoring their domain:

\[
S <: R \text{ if } \ldots \text{ and } U <: T \text{ and } \ldots
\]

Crucial observation.

The reason \( S <: R \) above must be false is as follows. Assume that the method \( m \) implemented by \( f : R \rightarrow T \) in \( R \) is overridden by a function \( f' : S \rightarrow T \) in \( S \) (which uses the additional components of \( S \)).

Now take \( r : R \) and \( s : S \); by subsumption \( s : R \), hence \( s.m(r) \) is legal.

But \( s.m = f' \), and \( r \) does not have the components \( f' \) requires.

However \( s.m(r) \) cannot ever happen, if we use the method abbreviation consistently (see above). If we respect this convention, we can safely use a covariant rule for methods, and as a consequence we obtain \( S <: R \), as desired.

Method: Lumens of a point

Let Rec LPoint =

\[
\{ \text{Point} <= \text{method lumens: LPoint\rightarrow Real} \}
\]

let makeLPoint(p: Point): LPoint =

\[
\{ p <= \\
\quad \text{method lumens(self: LPoint): Real = pixelLumens} \\
\};
\]

Let LDisc =

\[
\{ \text{LPoint} <= r : \text{Int} \}
\]

Note: LDisc is well-formed, and LDisc <: LPoint (non-trivial), since LDisc has the class-form:

\[
\{ \ldots \text{method lumens: LPoint\rightarrow Real} \ldots \} \quad \text{for } \text{LDisc} <: \text{LPoint}
\]
Inheriting a method

let makeLDisc(p: LPoint, r: Int): LDisc = {p | r=r}
makeLDisc(p, 0): LDisc

Overriding a method:

let makeLDisc(p: LPoint, r: Int): LDisc = {p | r=r}

Note: overriding lumens preserves the class-form of LDisc.

Inheriting a method

Let MDisc = {MPoint <- r: Int}
let makeMDisc(p: MPoint, r: Int): MDisc = {p | r=r}

But for d:MDisc, d<=incX: MPoint

Problem: the type of the incX method is not generic enough.

Method: Moving a point

Recall: the generic increment-X function:
let incX(R<:Point, r:R): {R<:x:Int} = {r <- x=r.x+1}

Let Rec MPoint = {Point <- method incX: MPoint->MPoint}
let makeMPoint(p: Point): MPoint = {p
  <- method incX(self: MPoint): MPoint = incX(MPoint, self) <- specialized incX
}

For d:MDisc, d<=incX: MPoint

Problem: this works, but we had to override a method just to change its type: failure of genericity.

Overriding a method:

Let MDisc = {MPoint <- method incX: MDisc->MDisc <- r: Int}
let makeMDisc(p: MPoint, r: Int): MDisc = {p
  <- method incX(self: MDisc): MDisc = incX(MDisc, self)
  | r=r}

For d:MDisc, d<=incX: MDisc

Problem: this works, but we had to override a method just to change its type: failure of genericity.
Non-generic Classes
A class is a record type in class-form:
\[ R = \{ \ldots \text{method } m : R' \rightarrow T \ldots \} \quad \text{for } R <: R' \]

A subclass is a subtype of a class, according to the method subtyping rule:
\[ R = \{ \ldots \text{method } m : R' \rightarrow T \ldots \} \quad \text{for } R <: R' \]
\[ S = \{ \ldots \text{method } m : S' \rightarrow U \ldots \} \quad \text{for } S <: S' \]
\[ S <: R \quad \text{if} \ldots \text{and } U <: T \text{ and } \ldots \]
Note that we are using structural subtyping, so that subclasses do not have to be "declared".

Because of structural subtyping, we implicitly obtain multiple inheritance.

To build a class C inheriting from two superclasses A and B, we could allow the syntax:

Let \( C = A \) and \( B \)

as an abbreviation for:

Let \( C = \{ A \leftarrow b_1 \ldots \leftarrow b_n \} \)

Note however that this is admissible if \( B \) is completely "known" to be \( \{ b_1, \ldots, b_n \} \) (which is the common case), but not in general if \( B \) has the form \( \{ B'| b_1 \ldots | b_n \} \) where \( B' \) is a (lambda-abstracted) type variable.

Generic Methods
The notion of methods considered so far does not extend nicely to methods of the form \( m: R \rightarrow R \), as we have seen in the incX example; such methods must be parametric, so that when they are inherited by a subclass \( S <: R \), they will have type \( m: S \rightarrow S \), and not \( m: R \rightarrow R \).

Hence methods should be generic:

\[ \text{for } r: R, \quad r \leq m \equiv r.m(R, r) \]

This requires modifying our class-form:
\[ R = \{ \ldots \text{method } m : \text{Fun}(S <: R', s:S) T \ldots \} \quad \text{for } R <: R' \]

Subtyping ignores \( R' \).
Let Rec MPoint =
{Point <-
  method incX(R:<MPoint, r:R):{R<-x:Int}
}

let p: MPoint =
{y=0,
  method incX(R:<MPoint, r:R):{R<-x:Int} =
  {r <- x=r.x+1}}

p<=incX (≡ p.incX(MPoint, p)) : MPoint

Let MColorPoint =
{MPoint <- rgb:RGB}

Inheriting a method

let cp: MColorPoint = {p <- rgb=red}
cp<=incX (≡ cp.incX(MColorPoint, cp)) : MColorPoint

(Finally!)

Overriding a method

let cp1: MColorPoint =
{p <- method incX(R:<MColorPoint, r:R):
  {R<-x:Int<-rgb:RGB} =
  {r <- x=r.x+1 <- rgb=invert(r.rgb)}
  <- rgb=red}
cp1<=incX (≡ cp1.incX(MColorPoint, cp1)) : MColorPoint

Generic Classes

A class is a record type in class-form:

R = {... method m: Fun(S:<R', s:S) T ...} for R:<R'

A subclass is a subtype of a class.

Nothing else is special.

CONCLUSIONS

Is this too complicated?

Maybe, but we think this just goes to shows that basic ideas in o-o languages are complex (how many people do you know who agree on what o-o languages are?).

Complex ideas are the ones that most need formal treatment. We are giving it a try.
APPENDIX: Type Rules

**Judgements**

- \( \vdash E \text{ env} \) (E is an environment)
- \( \vdash E \cdot K \) (K is a kind)
- \( \vdash E \cdot A \cdot \pi \) (A is a type)
- \( \vdash E \cdot A \cdot \alpha \) (A has kind \( \alpha \))

**Equivalence of kinds and types**

(Omitted. It involves reflativity, transitivity, congruence, typed \( \beta \)-conversion of type operators, and expansion of recursive types.)

**Conversion**

- \( \vdash E \cdot A \cdot \alpha \cdot \pi \cdot b \)
- \( \vdash E \cdot a \cdot b \)

**Inclusion**

- \( \vdash E \cdot K \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \)
- \( \vdash E \cdot A \cdot \alpha \cdot \pi \cdot b \)
- \( \vdash E \cdot a \cdot b \)

**Subsumption**

- \( \vdash E \cdot A \cdot \pi \cdot b \)
- \( \vdash E \cdot a \cdot b \)

**Environments**

- \( \vdash E \cdot K \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot x \cdot b \)

**The kind of types**

- \( \vdash E \cdot K \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot x \cdot b \)

**The kind of operators**

- \( \vdash E \cdot K \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \)
- \( \vdash E \cdot x \cdot b \)

**The type of polymorphic functions**

- \( \vdash E \cdot K \cdot K \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot x \cdot b \cdot b \)

**The type of functions**

- \( \vdash E \cdot A \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot x \cdot b \cdot b \)

**The type of data abstractions**

- \( \vdash E \cdot K \cdot K \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot x \cdot b \cdot b \)

**The type of pairs**

- \( \vdash E \cdot A \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot A \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot X \cdot K \cdot \pi \cdot b \cdot b \)
- \( \vdash E \cdot x \cdot b \cdot b \)
Record kinds

A countable set of labels \( u \), \( \ldots, u_n \) with finite domain \( \{u\} \), the kind of functions in \( \text{REC} \) which are defined over \( u \) and not defined over \( v \).

\[
\begin{align*}
\text{Formation} & : \\
\text{Subtyping} & : \\
\text{Equivalence} & :
\end{align*}
\]

\[
\begin{align*}
\text{Formation} & : \\
\end{align*}
\]

\[
\begin{align*}
\text{Subtyping} & : \\
\end{align*}
\]

\[
\begin{align*}
\text{Equivalence} & :
\end{align*}
\]

\[
\begin{align*}
\text{Introduction} & : \\
\text{Elimination} & : \\
\text{Reduction} & :
\end{align*}
\]