OOPSLA '88 Tutorial

Semantic Methods for Object-Oriented Languages

Part 2 of 3

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Ave, Palo Alto, CA 94301

John Mitchell

Department of Computer Science, Stanford University
Stanford, CA 94305

INTRODUCTION

OUTLINE

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

Record Types and Subtypes

Subtyping and Genericity

Methods (and Classes)

Generic Methods (and Classes)

PART 1

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}`	≅ `{a=3, b=true}`
`{a=3, b=true}\b`	≅ `{a=3}`
`{a=3, b=true}\c`	≅ `{a=3, b=true}`
`{{a=3} <- a=true}`	≅ `{{a=3}\a | a=true}`
`{a=3, b=true}.a`	≅ `3`

```
{a=3, a=4}
{{a=3}| a=4}
{a=3}.b
```


Record types

`:{a:Int, b:Bool}`	records with <u>at least</u> some fields (`{a=3, b=true, c="ab"}`, `{a=3}`)
`:{a:Int} \b`	records <u>without</u> some fields (`{a=3, c="ab"}`, `{a=3, b=true}`)
`:{}`	<u>all</u> records (`{}`, `{a=3}`, etc.)
`:{b:Bool, a:Int}`	≅ `:{a:Int, b:Bool}`
`:{a:Int, b:Bool}\b`	≅ `:{a:Int}\b`
`:{{a:Int}\b| b:Bool}`	≅ `:{a:Int, b:Bool}`
`:{{a:Int} <- a:Bool}`	≅ `:{{a:Int}\a | a:Bool}`

```
:{a:Int, a:Bool}
:{{a:Int} | a:Bool}
:{{a:Int} | b:Bool}
```


Record kinds

`::TYPE` the kind of all types

`::{}` the kind of all record types (e.g. `{a b:Int}\c`, `Int`)

`::{a,b}` the kind of all record types that have some components (e.g. `{a b:Int}\c`, `{a:Int}`)

`::{}\a b` the kind of those record types that do not have **a** or **b** components (e.g. `{c:Int}\a b`)

Record value variables

```
{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:

```
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

```
:{R| c:Bool}              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}
```

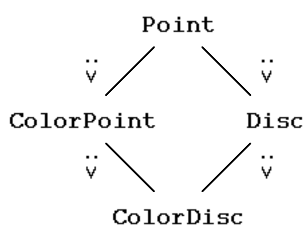
But where is **R** (which must not have **a** or **b**) bound? The information about what **R** must and must not have is "remembered" in its kind:

```
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}

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

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

Applicative style:

```
let incXofPoint(p: Point): Point =
     {p <- x=p.x+1}

let incXofCPoint(cp: ColorPoint): ColorPoint =
     {cp <- x=cp.x+1}
```


Pattern-matching style:

```
let incXofPoint({x=px, y=py}: Point): Point =
     {x=px+1, y=py}

let incXofPoint ({r| x=px}: Point): Point =
     {r| x=px+1}
```

Imperative style:

```
let incXofPoint (p: Point): Point =
     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 Point = {x y: Int}
Let ColorPoint = {Point <- rgb: RGB}
Let Disc = {Point <- r: Int}
Let ColorDisc = {ColorPoint <- r: Int}

let lumensOfPoint(p: Point): Real =
     pixelLumens
let lumensOfCPoint(cp: ColorPoint): Real =
     lumensOfPoint(cp) * rgbIntensity(cp.rgb)
let lumensOfDisc(d: Disc): Real =
     lumensOfPoint(d) * pi * d.r * d.r
let lumensOfCDisc(cd: ColorDisc): Real =
either   lumensOfCPoint(cd) * pi * cd.r * cd.r
or       lumensOfDisc(cd) * rgbIntensity(cd.rgb)
```


Record subtyping

```
Let Point = {x y: Int}
Let ColorPoint = {Point <- rgb: RGB}
```

then: **ColorPoint <: Point** subtyping

because **ColorPoint** has all the attributes of **Point**. This is subtyping in "width".

So, **Point** is the type of all the records which have at least **x** and **y** components.

Now, **Point\r** is the type of all the records which have at least **x** and **y**, but no **r**.

hence: **Point\r <: Point**

Structural subtyping:

Whether:

```
Let Point = {x y: Int}
Let ColorPoint = {Point <- rgb: RGB}
```

or:

```
Let Point = {x y: Int}
Let ColorPoint = {x y: 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 <- r: Int}
Let ColorDisc = {ColorPoint <- r: Int}
```


PART 2

SUBTYPING AND GENERICITY

Subsumption

If **a:A** and **A<:B** then **a:B**

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

let p: Point = {x=0, y=0}
let cp: ColorPoint = {p <- rgb=red}
let d: Disc = {p <- r=1}
let cd: ColorDisc = {cp <- r=1}
```

Then, by subsumption:

```
cp: Point
d: Point
cd: Disc, cd: ColorPoint, cd: Point
```

Also:

```
cp\r: Point\r
```

Loss of genericity

The crude use of subsumption causes loss of genericity:

Making a disc:

```
let discOfPoint(p: Point\r): Disc =
     {p| r=1}
```

Then **discOfPoint(cp\r)** is legal, by subsumption.
Note however that **discOfPoint(cp): Disc**.

Moving a point:

```
let incXofPoint(p: Point): Point =
     {p <- x=p.x+1}
```

Then **incXofPoint(cp)** is legal, by subsumption.
Note however that **incXofPoint(cp): 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:

```
let f(R::{}\x, r:{R|x:Int}): {R|x:Int} = ...
```

We shall also allow to abstract over all the subtypes of a given type, by bounded quantification:

```
let f(R<:Point, r:R): {R<-x:Int} = ...
```

(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 **Disc** when given a **Point**, and a **ColorDisc** when given a **ColorPoint**.

```
let discfy(P<:Point, p:P): {P<-r:Int} =
     {p<- r=1}

discfy(Point, p)
     : {Point<-r:Int} ≡ Disk

discfy(ColorPoint, cp)
     : {ColorPoint<-r:Int} ≡ ColorDisc
```

(Another possibility:

```
Let discfy(P<:Point\r, p:P): {P| r:Int} =
     {p| r=1}
discfy(Point\r, p\r)
```

)

Moving a point, generically

We want an increment-x function that returns a **Point** when given a **Point**, a **ColorPoint** when given a **ColorPoint**, etc.

```
let incX(R<:Point, r:R): {R<-x:Int} =
     {r <- x=r.x+1}

incX(Point, p): Point
incX(ColorPoint, cp): ColorPoint
incX(Disc, p): Disc
incX(ColorDisc, cp): ColorDisc
```

Note: why not

```
let incX(R<:Point, r:R): R =
     {r <- x=r.x+1}
```

Take `[0..9]<:Int` so that:

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

`incX(PointLT10, 9)` does not have type `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.

PART 3

METHODS (AND CLASSES)

Contravariance

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

```
f: T->ColorPoint  ⇒  f: T->Point
```

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

```
f: Point->U  ⇒  f: Disc->U
```

Combining the two:

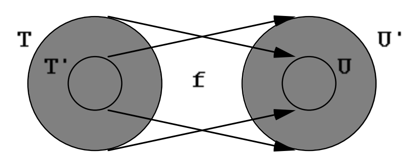
```
f: Point->ColorPoint  ⇒  f:Disc->Point
```

Therefore:

```
Point->ColorPoint <: Disc->Point
```


In general:

```
T'<:T  U<:U'
-------------
T->U <: T'->U'
```



Non-generic Methods

General idea: sending a message **m** to an object **r**.

for **r:R**, **r<=m** ≡ **r.m(r)**

where **m: R -> T** is a method (its first parameter is self).
(A common case: **T = R**.)
Hence **R = {... m: R->T ...}**.

Now consider a subtype **S<:R**, then we should have:

for **s:S**, **s<=m** ≡ **s.m(s)**

where **m: S -> T**. Hence **S = {... m: S->T ...}**.

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

Crucial observation.

The reason **S<:R** above must be false is as follows. Assume that the method **m** implemented by **f:R->T** in **R** is overridden by a function **f':S->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.

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

R = {... method m: R'->T ...} for **R<:R'**
S = {... method m: S'->U ...} for **S<:S'**

An object is a record with methods.

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

r <= m (≡ **r.m(r)**)

Methods have a special subtyping rule, ignoring their domain:

S<:R if ... and **U<:T** and ...

Method: Lumens of a point

```
Let Rec LPoint =
    {Point <- method lumens: LPoint->Real}

let makeLPoint(p: Point): LPoint =
    {p <-
      method lumens(self: LPoint): Real =
        pixelLumens
    }

Let LDisc =
    {LPoint <- r: Int}
```

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

LDisc = {... method lumens: LPoint->Real ...}
for **LDisc <: LPoint**

Inheriting a method

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

makeLDisc(p\r, 0): LDisc
```

Overriding a method:

```
let makeLDisc(p: LPoint\r, r: Int): LDisc =
     {p
      <-    lumens(self: LDisc): Real =
                 pixelLumens*pi*self.r*self.r
      |   r=r
     }
```

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

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
     }
```


Inheriting a method

```
Let MDisc =
     {MPoint <- r: Int}

let makeMDisc(p: MPoint\r, r: Int): MDisc =
     {p| r=r}
```

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

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

Overriding a method:

```
Let MDisc =
     {MPoint
        <- method incX: MDisc->MDisc
        <- r: Int}

let makeMDisc(p: MPoint\r, 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 = {... method m: R'->T ...} for **R<:R'**

A subclass is a subtype of a class, according to the method subtyping rule:

R = {... method m: R'->T ...} for **R<:R'**
S = {... method m: S'->U ...} for **S<:S'**

S<:R if ... and **U<:T** and ...

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 <- b_1 ... <- b_n}

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

PART 4

GENERIC METHODS (AND CLASSES)

Generic Methods

The notion of methods considered so far does not extend nicely to methods of the form **m: R -> 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 -> S**, and not **m: R -> R**.

Hence methods should be generic:

for **r:R,** **r <= m** ≡ **r.m(R, r)**

This requires modifying our class-form:

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

Subtyping ignores **R'**.

```
Let Rec MPoint =
   {Point <-
    method incX(R<:MPoint, r:R):{R<-x:Int}
   }

let p: MPoint =
   {x=0, 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		
$E \vdash K \text{ kind}$	K is a kind	$E \vdash A::K$	A has kind K
$E \vdash A \text{ type}$	A is a type (same as $E \vdash A::\mathcal{T}$)	$E \vdash a:A$	a has type A
$E \vdash K<::L$	K is a subkind of L	$E \vdash K<::>L$	equivalent kinds
$E \vdash A<:B$	A is a subtype of B	$E \vdash A<:>B$	equivalent types

Equivalence of kinds and types

(Omitted. It involves reflexivity, transitivity, congruence, typed β-conversion of type operators, and expansion of recursive types.)

Conversion

$$\frac{E \vdash A::K \quad E \vdash K<::>L}{E \vdash A :: L} \qquad \frac{E \vdash a:A \quad E \vdash A<:>B}{E \vdash a : B}$$

Inclusion

$$\frac{E \vdash K \text{ kind}}{E \vdash K <:: K} \qquad \frac{E \vdash A \text{ type}}{E \vdash A <: A}$$

$$\frac{E \vdash K<::L \quad E \vdash L<::M}{E \vdash K <:: M} \qquad \frac{E \vdash A<:B \quad E \vdash B<:C}{E \vdash A <: C}$$

Subsumption

$$\frac{E \vdash A::K \quad E \vdash K<::L}{E \vdash A<::L} \qquad \frac{E \vdash a:A \quad E \vdash A<:B}{E \vdash a : B}$$

Environments

$$\vdash \varnothing \text{ env} \qquad \frac{E \vdash K \text{ kind} \quad X \notin Dom(E)}{\vdash E, X::K \text{ env}} \qquad \frac{E \vdash A \text{ type} \quad x \notin Dom(E)}{\vdash E, x:A \text{ env}}$$

Variables

$$\frac{\vdash E \text{ env} \quad X \in Dom(E)}{E \vdash X :: E(X)} \qquad \frac{\vdash E \text{ env} \quad x \in Dom(E)}{E \vdash x : E(x)}$$

The kind of types

$$\frac{\vdash E \text{ env}}{E \vdash \mathcal{T} \text{ kind}}$$

The kind of operators

$$\frac{E \vdash K \text{ kind} \quad E, X::K \vdash L \text{ kind}}{E \vdash \Pi(X::K)L \text{ kind}}$$

$$\frac{E \vdash K \text{ kind} \quad E, X::K \vdash B::L}{E \vdash \lambda(X::K)B :: \Pi(X::K)L}$$

$$\frac{E \vdash B::\Pi(X::K)L \quad E \vdash A::K}{E \vdash B(A) :: L\{X \leftarrow A\}}$$

$$\frac{E \vdash K'<::K \quad E, X::K' \vdash L<::L'}{E \vdash \Pi(X::K)L <:: \Pi(X::K')L'}$$

The type of polymorphic functions

$$\frac{E \vdash K \text{ kind} \quad E, X::K \vdash B \text{ type}}{E \vdash \Pi(X::K)B \text{ type}}$$

$$\frac{E \vdash K \text{ kind} \quad E, X::K \vdash b:B}{E \vdash \lambda(X::K)b : \Pi(X::K)B}$$

$$\frac{E \vdash b:\Pi(X::K)B \quad E \vdash A::K}{E \vdash b(A) : B\{X \leftarrow A\}}$$

$$\frac{E \vdash K'<::K \quad E, X::K' \vdash B<:B'}{E \vdash \Pi(X::K)B <: \Pi(X::K')B'}$$

The type of functions

$$\frac{E \vdash A \text{ type} \quad E \vdash B \text{ type}}{E \vdash A \to B \text{ type}}$$

$$\frac{E \vdash A \text{ type} \quad E, x:A \vdash b:B}{E \vdash \lambda(x:A)b : A \to B}$$

$$\frac{E \vdash b:A \to B \quad E \vdash a:A}{E \vdash b(a) : B}$$

$$\frac{E \vdash A'<:A \quad E \vdash B<:B'}{E \vdash A \to B <: A' \to B'}$$

The type of data abstractions

$$\frac{E \vdash K \text{ kind} \quad E, X::K \vdash B \text{ type}}{E \vdash \Sigma(X::K)B \text{ type}}$$

$$\frac{E \vdash A::K \quad E \vdash b\{X \leftarrow A\}:B\{X \leftarrow A\}}{E \vdash \langle X::K=A, b\rangle : \Sigma(X::K)B}$$

$$\frac{E \vdash c:\Sigma(X::K)B \quad E \vdash D \text{ type} \quad E, X::K, y:B \vdash d:D}{E \vdash \text{let}\,\langle X::K, y:B\rangle = c \text{ in } d : D}$$

$$\frac{E \vdash K<::K' \quad E, X::K \vdash B<:B'}{E \vdash \Sigma(X::K)B <: \Sigma(X::K')B'}$$

The type of pairs

$$\frac{E \vdash A \text{ type} \quad E \vdash B \text{ type}}{E \vdash A \times B \text{ type}}$$

$$\frac{E \vdash a:A \quad E \vdash b\{x \leftarrow a\}:B}{E \vdash \langle x:A=a, b\rangle : A \times B}$$

$$\frac{E \vdash c:A \times B \quad E, x:A, y:B \vdash d:D}{E \vdash \text{let}\,\langle x:A, y:B\rangle = c \text{ in } d : D}$$

$$\frac{E \vdash A<:A' \quad E \vdash B<:B'}{E \vdash A \times B <: A' \times B'}$$

Record kinds

U	≡	a countable set of labels $u \in U$
u	≡	$u_1 \dots u_n$
RECORD	≡	the kindof functions in $U \to TYPE$ with finite domain
{u}\v	≡	the kind of functions in RECORD which are defined over u and *not* defined over v

$$\frac{\vdash E\ \text{env}}{E \vdash \{u\}\backslash v\ \text{kind}} \qquad \frac{\vdash E\ \text{env}}{E \vdash \{uu\}\backslash v <:: \{u\}\backslash v} \qquad \frac{\vdash E\ \text{env}}{E \vdash \{u\}\backslash vv <:: \{u\}\backslash v}$$

$$\frac{\vdash E\ \text{env}}{E \vdash \{uuu\}\backslash v <::> \{uu\}\backslash v} \qquad \frac{\vdash E\ \text{env}}{E \vdash \{u\}\backslash vvv <::> \{u\}\backslash vv} \qquad \frac{\vdash E\ \text{env}}{E \vdash \{uu\}\backslash v <::> \{uu\}\backslash v} \qquad \frac{\vdash E\ \text{env}}{E \vdash \{u\}\backslash vv <::> \{u\}\backslash vv}$$

The type of records

Formation

$$\frac{\vdash E\ \text{env}}{E \vdash \{\} :: \{\}} \qquad \frac{E \vdash R::\{u\}\backslash vv \quad E \vdash A\ \text{type}}{E \vdash \{R|\ v{:}A\} :: \{uv\}\backslash v} \qquad \frac{E \vdash R::\{uu\}\backslash v}{E \vdash R\backslash u ::\ \{u\}\backslash vu} \qquad \frac{E \vdash R::\{u\}\backslash v}{E \vdash R\backslash u ::\ \{u\}\backslash vu}$$

Subtyping

$$\frac{E \vdash R,R'::\{u\}\backslash vv \quad E \vdash R<:R' \quad E \vdash A<:A'}{E \vdash \{R|\ v{:}A\}\ <:\ \{R'|\ v{:}A'\}} \qquad \frac{E \vdash R::\{u\}\backslash vv \quad E \vdash R<:R' \quad E \vdash A\ \text{type}}{E \vdash \{R|\ v{:}A\}\ <:\ R'}$$

$$\frac{E \vdash R<:R'}{E \vdash R\backslash u\ <:\ R'\backslash u} \qquad \frac{\vdash E\ \text{env}}{E \vdash \{\}\backslash u <:\ \{\}}$$

Equivalence

$$\frac{\vdash E\ \text{env}}{E \vdash \{\} <:> \{\}} \qquad \frac{E \vdash R::\{u\}\backslash vvv' \quad E \vdash A,A'::\text{type}}{E \vdash \{\{R|\ v{:}A\}|\ v'{:}A'\}\ <:>\ \{\{R|\ v'{:}A'\}|\ v{:}A\}}$$

$$\frac{E \vdash R::\{u\}\backslash vv}{E \vdash R\backslash v <:>\ R} \qquad \frac{E \vdash R::\{u\}\backslash v}{E \vdash R\backslash uu' <:>\ R\backslash u'u}$$

$$\frac{E \vdash R::\{u\}\backslash vv}{E \vdash \{R|\ v{:}A\}\backslash v <:> R} \qquad \frac{E \vdash R::\{u\}\backslash vv \quad E \vdash A::\text{type} \quad u{\neq}v}{E \vdash \{R|\ v{:}A\}\backslash u\ <:>\ \{R\backslash u|\ v{:}A'\}}$$

Introduction

$$\frac{\vdash E\ \text{env}}{E \vdash \{\}\ :\ \{\}\backslash u} \qquad \frac{E \vdash r{:}R::\{u\}\backslash vv \quad E \vdash a{:}A}{E \vdash \{r|\ v{=}a\} : \{R|\ v{:}A\}}$$

Elimination

$$\frac{E \vdash r{:}R}{E \vdash r\backslash u : R\backslash u} \qquad \frac{E \vdash r{:}\{R|\ u{:}A\}}{E \vdash r.u : A}$$

Reduction

$$\frac{\vdash E\ \text{env}}{E \vdash \{\} \leftrightarrow \{\}} \qquad \frac{E \vdash r{:}R::\{u\}\backslash vvv' \quad E \vdash a{:}A \quad E \vdash a'{:}A' \quad v{\neq}v'}{E \vdash \{\{r|\ v{=}a\}|\ v'{=}a''\}\ \leftrightarrow\ \{\{r|\ v'{=}a''\}|\ v{=}a\}} \qquad \frac{E \vdash r{:}R::\{u\}\backslash vv}{E \vdash \{r|\ v{=}a\}.v\ \leftrightarrow\ a}$$

$$\frac{\vdash E\ \text{env}}{E \vdash\ \{\}\backslash u \leftrightarrow\ \{\}} \qquad \frac{E \vdash r{:}R}{E \vdash r\backslash uu \leftrightarrow\ r\backslash u} \qquad \frac{E \vdash r{:}R}{E \vdash r\backslash uu' \leftrightarrow\ r\backslash u'u}$$