Operationally Sound Update

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

Outline

- The type rules necessary for "sufficiently polymorphic" update operations on records and objects are based on unusual operational assumptions.
- These update rules are sound operationally, but not denotationally (in standard models). They arise naturally in type systems for programming, and are not easily avoidable.
- Thus, we have a situation where operational semantics is clearly more advantageous than denotational semantics.
- However (to please the semanticists) I will show how these operationally-based type systems can be translated into type systems that are denotationally sound.

The polymorphic update problem

L.Cardelli, P.Wegner

"*The need for bounded quantification arises very frequently in object-oriented programming. Suppose we have the following types and functions:*

type *Point* = [*x*: *Int*, *y*: *Int*]
value $moveX_0$ = λ(*p*: *Point*, *dx*: *Int*) *p.x* := *p.x* + *dx*; *p*
value *moveX* = λ(*P* <: *Point*) λ(*p*: *P*, *dx*: *Int*) *p.x* := *p.x* + *dx*; *p*

It is typical in (type-free) object-oriented programming to reuse functions like moveX on objects whose type was not known when moveX was defined. If we now define:

type *Tile* = [*x*: *Int*, *y*: *Int*, *hor*: *Int*, *ver*: *Int*]

we may want to use moveX to move tiles, not just points."

Tile <: *Point*

$moveX_0$([*x*=0, *y*=0, *hor*=1, *ver*=1], 1).*hor* fails

moveX(*Tile*)([*x*=0, *y*=0, *hor*=1, *ver*=1], 1).*hor* succeeds

- In that paper, bounded quantification was justified as a way of handling polymorphic update, and was used in the context of *imperative* update.
- The examples were inspired by object-oriented applications. Object-oriented languages combine subtyping and polymorphism with state encapsulation, and hence with imperative update. Some form of polymorphic update is inevitable.
- Simplifying the situation a bit, let's consider the equivalent example in a functional setting. We might hope to achieve the following typing:

  bump ≜ λ(*P* <: *Point*) λ(*p*: *P*) *p.x* := *p.x* + 1

  bump : ∀(*P* <: *Point*) *P*→*P*

  But ...

There is no bump there!

Neither semantically

J.Mitchell

In standard models, the type $\forall(P<:Point)P\to P$ contains only the identity function.

Consider $\{p\}$ for any $p\in Point$. If $f:\forall(P<:Point)P\to P$, then $f(\{p\}):\{p\}\to\{p\}$, therefore f must map every point to itself, and must be the identity.

Nor parametrically

M.Abadi, L.Cardelli, G.Plotkin

By parametricity (for bounded quantifiers), we can show that if $f:\forall(P<:Point)P\to P$, then $\forall(P<:Point)\ \forall(x:P)\ f(P)(x) =_P x$. Thus f is an identity.

Nor by standard typing rules

As shown next ...

The simple rule for update

(Val Simple Update)

$$\frac{E\vdash a : [l_i{:}B_i\ ^{i\in 1..n}] \qquad E\vdash b : B_j \qquad j\in 1..n}{E\vdash a.l_j{:=}b : [l_i{:}B_i\ ^{i\in 1..n}]}$$

- According to this rule, bump does not typecheck as desired:

  $bump \triangleq \lambda(P <: Point)\ \lambda(p: P)\ p.x := p.x + 1$

  We must go from $p{:}P$ to $p{:}Point$ by subsumption before we can apply the rule. Therefore we obtain only:

  $bump : \forall(P <: Point)\ P\to Point$

The "structural" rule for update

(Val Structural Update)

$$\frac{E\vdash a : A \qquad E\vdash A <: [l_i{:}B_i\ ^{i\in 1..n}] \qquad E\vdash b : B_j \qquad j\in 1..n}{E\vdash a.l_j{:=}b : A}$$

- According to this rule, bump typechecks as desired, using the special case where A is a type variable.

  $bump \triangleq \lambda(P <: Point)\ \lambda(p: P)\ p.x := p.x + 1$

  $bump : \forall(P <: Point)\ P\to P$

- Therefore, (Val Structural Update) is not sound in most semantic models, because it populates the type $\forall(P<:Point)P\to P$ with a non-identity function.

- However, (Val Structural Update) is in practice highly desirable, so the interesting question is under which conditions it is sound.

Can't allow too many subtypes

- Suppose we had:

  $BoundedPoint \triangleq \{x: 0..9, y: 0..9\}$

  $BoundedPoint <: Point$

  then:

  $bump(BoundedPoint)(\{x{=}9, y{=}9\}) : BoundedPoint$

  unsound!

- To recover from this problem, the subtyping rule for records/objects must forbid certain subtypings:

  (Sub Object)

  $$\frac{E\vdash B_i \qquad \forall i\in 1..m}{E\vdash [l_i{:}B_i\ ^{i\in 1..n+m}] <: [l_i{:}B_i\ ^{i\in 1..n}]}$$

- Therefore, for soundness, the rule for structural updates makes implicit assumptions about the subtype relationships that may exist.

Relevant rules for structural update

(Sub Object)

$$\frac{E \vdash B_i \qquad \forall i \in 1..m}{E \vdash [l_i{:}B_i\ ^{i\in 1..n+m}] <: [l_i{:}B_i\ ^{i\in 1..n}]}$$

(Val Subsumption)

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

(Val Object)

$$\frac{E \vdash b_i : B_i \qquad \forall i \in 1..n}{E \vdash [l_i{=}b_i\ ^{i\in 1..n}] : [l_i{:}B_i\ ^{i\in 1..n}]}$$

(Val Structural Update)

$$\frac{E \vdash a : A \qquad E \vdash A <: [l_i{:}B_i\ ^{i\in 1..n}] \qquad E \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j{:=}b : A}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i{=}v_i\ ^{i\in 1..n}] \qquad \vdash b \rightsquigarrow v \qquad j \in 1..n}{\vdash a.l_j{:=}b \rightsquigarrow [l_j{=}v,\ l_i{=}v_i\ ^{i\in 1..n-\{j\}}]}$$

The structural subtyping lemmas

Lemma (Structural subtyping)

If $E \vdash [l_i{:}B_i\ ^{i\in I}] <: C$ then either $C \equiv Top$, or $C \equiv [l_i{:}B_i\ ^{i\in J}]$ with $J \subseteq I$.

If $E \vdash C <: [l_i{:}B_i\ ^{i\in J}]$ then either $C \equiv [l_i{:}B_i\ ^{i\in I}]$ with $J \subseteq I$,

or $C \equiv X_1$ and E contains a chain $X_1 <: \ldots <: X_p <: [l_i{:}B_i\ ^{i\in I}]$ with $J \subseteq I$.

Proof

By induction on the derivations of $E \vdash [l_i{:}B_i\ ^{i\in I}] <: C$ and $E \vdash C <: [l_i{:}B_i\ ^{i\in I}]$.

□

Soundness by subject reduction

Theorem (Subject reduction)

If $\emptyset \vdash a : A$ and $\vdash a \rightsquigarrow v$ then $\emptyset \vdash v : A$.

Proof By induction on the derivation of $\vdash a \rightsquigarrow v$.

Case (Red Update)

$$\frac{\vdash c \rightsquigarrow [l_i{=}z_i\ ^{i\in 1..n}] \qquad \vdash b \rightsquigarrow w \qquad j \in 1..n}{\vdash c.l_j{:=}b \rightsquigarrow [l_j{=}w,\ l_i{=}z_i\ ^{i\in 1..n-\{j\}}]}$$

By hypothesis $\emptyset \vdash c.l_j{:=}b : A$. This must have come from (1) an application of (Val Structural Update) with assumptions $\emptyset \vdash c : C$, and $\emptyset \vdash C <: D$ where $D \equiv [l_j{:}B_j, \ldots]$, and $\emptyset \vdash b : B_j$, and with conclusion $\emptyset \vdash c.l_j{:=}b : C$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C <: A$ by transitivity.

By induction hypothesis, since $\emptyset \vdash c : C$ and $\vdash c \rightsquigarrow z \equiv [l_i{=}z_i\ ^{i\in 1..n}]$, we have $\emptyset \vdash z : C$.

By induction hypothesis, since $\emptyset \vdash b : B_j$ and $\vdash b \rightsquigarrow w$, we have $\emptyset \vdash w : B_j$.

Now, $\emptyset \vdash z : C$ must have come from (1) an application of (Val Object) with assumptions $\emptyset \vdash z_i : B_i'$ and $C' \equiv [l_i'{:}B_i'\ ^{i\in 1..n}]$, and with conclusion $\emptyset \vdash z : C'$, followed by (2) a number of subsumption steps implying $\emptyset \vdash C' <: C$ by transitivity. By transitivity, $\emptyset \vdash C' <: D$. Hence by the Structural Subtyping Lemma, we must have $B_j \equiv B_j'$. Thus $\emptyset \vdash w : B_j'$. Then, by (Val Object), we obtain $\emptyset \vdash [l_j{=}w, l_i{=}z_i\ ^{i\in 1..n-\{j\}}] : C'$. Since $\emptyset \vdash C' <: A$ by transitivity, we have $\emptyset \vdash [l_j{=}w, l_i{=}z_i\ ^{i\in 1..n-\{j\}}] : A$ by subsumption.

Other structural rules

- Rules based on structural assumptions (structural rules, for short) are not restricted to record/object update. They also arise in:
  - ~ method invocation with Self types,
  - ~ object cloning,
  - ~ class encodings,
  - ~ unfolding recursive types.
- The following is one of the simplest examples of the phenomenon (although not very useful in itself):

A structural rule for product types

M.Abadi

- The following rule for pairing enables us to mix two pairs a and b of type C into a new pair of the same type. The only assumption on C is that it is a subtype of a product type $B_1 \times B_2$.

$$\frac{E \vdash C <: B_1 \times B_2 \qquad E \vdash a : C \qquad E \vdash b : C}{E \vdash \langle fst(a), snd(b) \rangle : C}$$

  The soundness of this rule depends on the property that every subtype of a product type $B_1 \times B_2$ is itself a product type $C_1 \times C_2$.
- This property is true operationally for particular systems, but fails in any semantic model where subtyping is interpreted as the subset relation. Such a model would allow the set $\{a,b\}$ as a subtype of $B_1 \times B_2$ whenever a and b are elements of $B_1 \times B_2$. If a and b are different, then $\langle fst(a), snd(b) \rangle$ is not an element of $\{a,b\}$. Note that $\{a,b\}$ is not a product type.

A structural rule for recursive types

M.Abadi, L.Cardelli, R.Viswanathan

- In the paper "An Interpretation of Objects and Object types" we give a translation of object types into ordinary types:

$$[l_i{:}B_i\ ^{i \in 1..n}] \triangleq \mu(Y)\ \exists(X<:Y)\ \langle r{:}X,\ l_i^{sel}{:}X \to B_i\ ^{i \in 1..n},\ l_i^{upd}{:}(X \to B_i) \to X\ ^{i \in 1..n} \rangle$$

  this works fine for non-structural rules.

- In order to validate a structural update rule in the source calculus, we need a structural update rule in the target calculus. It turns out that the necessary rule is the following, which is operationally sound:

$$\frac{E \vdash C <: \mu(X)B\{X\} \qquad E \vdash a : C}{E \vdash unfold(a) : B[\![C]\!]}$$

A structural rule for method invocation

- In the context of object types with Self types:

(Val Select)

$$\frac{E \vdash a : A \qquad E \vdash A <: Obj(X)[l_i{:}B_i\{X\}\ ^{i \in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j : B_j[\![A]\!]}$$

This structural rule is necessary to "encapsulate" structural update inside methods:

$A \triangleq Obj(X)[n{:}\ Int,\ bump{:}\ X]$

$\lambda(Y <: A)\ \lambda(y{:}\ Y)\ y.bump$
$:\ \forall(Y <: A)\ Y \to Y$

Structural rules and class encodings

Types of the form $\forall(X<:A)X \to B\{X\}$ are needed also for defining classes as collections of pre-methods. Each pre-method must work for all possible subclasses, parametrically in self, so that it can be inherited.

$A \triangleq Obj(X)[l_i{:}B_i\{X\}\ ^{i \in 1..n}]$

$Class(A) \triangleq [new{:}\ A,\ l_i{:}\ \forall(X<:A)X \to B_i\{X\}\ ^{i \in 1..n}]$

$Bump \triangleq Obj(X)[n{:}\ Int,\ bump{:}\ X]$

$Class(Bump) \triangleq [new{:}\ Bump,\ bump{:}\ \forall(X<:Bump)X \to X]$

$c : Class(Bump) \triangleq$
$[new = \varsigma(c{:}\ Class(Bump))\ [n = 0,\ bump = \varsigma(s{:}\ Bump)\ c.bump(Bump)(s)],$
$bump = \lambda(X<:Bump)\ \lambda(x{:}X)\ x.n{:=}x.n{+}1\}]$

A structural rule for cloning

- In the context of imperative object calculi:

(Val Clone)

$$\frac{E \vdash a : A \qquad E \vdash A <: [l_i{:}B_i\ ^{i\in 1..n}] \qquad j \in 1..n}{E \vdash \mathit{clone}(a) : A}$$

This structural rule is necessary for bumping and returning a clone instead of the original object:

$$\mathit{bump} \triangleq \lambda(P <: \mathit{Point})\ \lambda(p: P)\ \mathit{clone}(p).x := p.x + 1$$

$$\mathit{bump} : \forall(P <: \mathit{Point})\ P \to P$$

Comments

- Structural rules are quite satisfactory. The operational semantics is the right one, the typing rules are the right ones for writing useful programs, and the rules are sound for the semantics.
- We do not have a denotational semantics (yet?). (The paper "Operations on Records" by L.Cardelli and J.Mitchell contains a limited model for structural update; no general models seems to be known.)
- Even without a denotational semantcs, there is an operational semantics from which one could, hopefully, derive a theory of typed equality.
- Still, I would like to understand in what way a type like $\forall(X<:\text{Point})X\to X$ does not mean what most people in this room might think.
- Insight may come from translating a calculus with structural rules, into one without structural rules for which we have a standard semantics.

Translating away structural rules

- The "Penn translation" can be used to map $F_{<:}$ into F by threading *coercion* functions.
- Similarly, we can map an $F_{<:}$-like calculus with structural rules into a normal $F_{<:}$-like calculus by threading *update* functions (*c.f.* M.Hofmann and B.Pierce: Positive subtyping).
- Example :

$$f : \forall(X <: [l: \mathit{Int}])\ X \to X \triangleq \lambda(X <: [l: \mathit{Int}])\ \lambda(x: X)\ x.l := 3$$

$$f\ ([l: \mathit{Int}])$$

(N.B. the update $x.l$:=3 uses the structural rule)

translates to:

$$f : \forall(X <: [l: \mathit{Int}])\ [l: X\to \mathit{Int}\to X] \to X \to X \triangleq \lambda(X <: [l: \mathit{Int}])\ \lambda(\pi_X: [l: X\to \mathit{Int}\to X])\ \lambda(x: X)\ \pi_X.l(x)(3)$$

$$f\ ([l: \mathit{Int}])\ ([l = \lambda(x: [l: \mathit{Int}])\ \lambda(y: \mathit{Int})\ x.l := y])$$

(N.B. the update $x.l$:=y uses the non-structural rule)

- Next I discuss a simplified, somewhat ad-hoc, calculus to formalize the main ideas for this translation.

Syntax

A,B ::=	types
X	type variable
$[l_i{:}B_i\ ^{i\in 1..n}]$	object type (l_i distinct)
$A\to B$	function types
$\forall(X<:[l_i{:}B_i\ ^{i\in 1..n}])B$	bounded universal type
a,b ::=	terms
x	variable
$[l_i=\varsigma(x_i{:}A_i)b_i\ ^{i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l\Leftarrow\varsigma(x{:}A)b$	method update
$\lambda(x{:}A)b$	function
$b(a)$	application
$\lambda(X<:[l_i{:}B_i\ ^{i\in 1..n}])b$	polymorphic function
$b(A)$	polymorphic instantiation

- We consider method update instead of field update ($a_A.l{:=}b \triangleq a.l\Leftarrow\varsigma(x{:}A)b$).
- We do not consider object types with Self types.
- We do not consider arbitrary bounds for type variables, only object-type bounds.

Environments

(Env $\emptyset$) $\quad \dfrac{}{\emptyset \vdash \diamond}$

(Env x) $\quad \dfrac{E \vdash A \qquad x \notin dom(E)}{E, x{:}A \vdash \diamond}$

(Env $X{<:}$) $\quad \dfrac{E \vdash A \qquad X \notin dom(E)}{E, X{<:}A \vdash \diamond}$

Types

(Type $X{<:}$) $\quad \dfrac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$

(Type Object) (l_i distinct) $\quad \dfrac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i{:}B_i{}^{\,i \in 1..n}]}$

(Type Arrow) $\quad \dfrac{E \vdash A \qquad E \vdash B}{E \vdash A \to B}$

(Type All$<:$) $\quad \dfrac{E, X{<:}A \vdash B}{E \vdash \forall (X{<:}A) B}$

Subtyping

(Sub Refl) $\quad \dfrac{E \vdash A}{E \vdash A <: A}$

(Sub Trans) $\quad \dfrac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$

(Sub X) $\quad \dfrac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X{<:}A}$

(Sub Object) (l_i distinct) $\quad \dfrac{E \vdash B_i \qquad \forall i \in 1..n{+}m}{E \vdash [l_i{:}B_i{}^{\,i \in 1..n+m}] <: [l_i{:}B_i{}^{\,i \in 1..n}]}$

(Sub Arrow) $\quad \dfrac{E \vdash A' <: A \qquad E \vdash B <: B'}{E \vdash A \to B <: A' \to B'}$

(Sub All) $\quad \dfrac{E \vdash A' <: A \qquad E, X{<:}A' \vdash B <: B'}{E \vdash \forall (X{<:}A) B <: \forall (X{<:}A') B'}$

Typing

(Val Subsumption) $\quad \dfrac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$

(Val x) $\quad \dfrac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x{:}A}$

(Val Object) (where $A \equiv [l_i{:}B_i{}^{\,i \in 1..n}]$) $\quad \dfrac{E, x_i{:}A \vdash b_i : B_i \qquad \forall i \in 1..n}{E \vdash [l_i{=}\varsigma(x_i{:}A) b_i{}^{\,i \in 1..n}] : A}$

(Val Select) $\quad \dfrac{E \vdash a : [l_i{:}B_i{}^{\,i \in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j : B_j}$

(Val Update Obj) (where $A \equiv [l_i{:}B_i{}^{\,i \in 1..n}]$) $\quad \dfrac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A) b : A}$

(Val Update X) (where $A \equiv [l_i{:}B_i{}^{\,i \in 1..n}]$) $\quad \dfrac{E \vdash a : X \qquad E \vdash X{<:}A \qquad E, x{:}X \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}X) b : X}$

(Val Fun) $\quad \dfrac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A) b : A \to B}$

(Val Appl) $\quad \dfrac{E \vdash b : A \to B \qquad E \vdash a : A}{E \vdash b(a) : B}$

(Val Fun2$<:$) $\quad \dfrac{E, X{<:}A \vdash b : B}{E \vdash \lambda(X{<:}A) b : \forall (X{<:}A) B}$

(Val Appl2$<:$) (where $A' \equiv [l_i{:}B_i{}^{\,i \in 1..n}]$ or $A' \equiv Y$) $\quad \dfrac{E \vdash b : \forall (X{<:}A) B\{X\} \qquad E \vdash A' <: A}{E \vdash b(A') : B\{\!|A'|\!\}}$

- The source system for the translation is the one given above. The target system is the one given above minus the (Val Update X) rule.
- Derivations in the source system can be translated to derivations that do not use (Val Update X). The following tables give a slightly informal summary of the translation on derivations.

Translation of Environments

$$\langle\!|\emptyset|\!\rangle \triangleq \emptyset$$

$$\langle\!|E, x{:}A|\!\rangle \triangleq \langle\!|E|\!\rangle, x{:}\langle\!|A|\!\rangle$$

$$\langle\!|E, X{<:}[l_i{:}B_i{}^{\,i \in 1..n}]|\!\rangle \triangleq \langle\!|E|\!\rangle, X{<:}\langle\!|[l_i{:}B_i{}^{\,i \in 1..n}]|\!\rangle, \pi_X{:}[l_i{:}X \to (X \to \langle\!|B_i|\!\rangle) \to X{}^{\,i \in 1..n}]$$

where each l_i: $X \to (X \to \langle\!|B_i|\!\rangle) \to X$ is an updator that takes an object of type X, takes a pre-method for X (of type $X \to \langle\!|B_i|\!\rangle$), updates the i-th method of the object, and returns the modified object of type X.

Translation of Types

$$\langle\!\langle X \rangle\!\rangle \triangleq X$$

$$\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle \triangleq [l_i{:}\langle\!\langle B_i \rangle\!\rangle\ ^{i\in 1..n}]$$

$$\langle\!\langle A{\to}B \rangle\!\rangle \triangleq \langle\!\langle A \rangle\!\rangle{\to}\langle\!\langle B \rangle\!\rangle$$

$$\langle\!\langle \forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])B \rangle\!\rangle \triangleq \forall(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle)[l_i{:}X{\to}(X{\to}\langle\!\langle B_i \rangle\!\rangle){\to}X\ ^{i\in 1..n}]{\to}\langle\!\langle B \rangle\!\rangle$$

- N.B. the translation preserves subtyping. In particular:

$$\langle\!\langle \forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])B \rangle\!\rangle <: \langle\!\langle \forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n+m}])B \rangle\!\rangle$$

since:

$$\forall(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle)\ [l_i{:}X{\to}(X{\to}\langle\!\langle B_i \rangle\!\rangle){\to}X\ ^{i\in 1..n}]{\to}\langle\!\langle B \rangle\!\rangle <:$$
$$\forall(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n+m}] \rangle\!\rangle)\ [l_i{:}X{\to}(X{\to}\langle\!\langle B_i \rangle\!\rangle){\to}X\ ^{i\in 1..n+m}]{\to}\langle\!\langle B \rangle\!\rangle$$

- We have a calculus with polymorphic update where quantifier and arrow types are contravariant on the left (*c.f.* Positive Subtyping).

Translation of Terms

$$\langle\!\langle x \rangle\!\rangle \triangleq x$$

$$\langle\!\langle [l_i{=}(x_i{:}A_i)b_i\ ^{i\in 1..n}] \rangle\!\rangle \triangleq [l_i{=}\varsigma(x_i{:}\langle\!\langle A_i \rangle\!\rangle)\langle\!\langle b_i \rangle\!\rangle\ ^{i\in 1..n}]$$

$$\langle\!\langle a.l_j \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l_j$$

$$\langle\!\langle a.l \Leftarrow \varsigma(x{:}A)b \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l \Leftarrow (x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle) \quad \text{for (Val Update Obj)}$$

$$\langle\!\langle a.l \Leftarrow \varsigma(x{:}X)b \rangle\!\rangle \triangleq \pi_X.l(\langle\!\langle a \rangle\!\rangle)(\lambda(x{:}X)\langle\!\langle b \rangle\!\rangle) \quad \text{for (Val Update } X\text{)}$$

$$\langle\!\langle \lambda(x{:}\mathrm{A})b \rangle\!\rangle \triangleq \lambda(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle$$

$$\langle\!\langle b(a) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle(\langle\!\langle a \rangle\!\rangle)$$

$$\langle\!\langle \lambda(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])b \rangle\!\rangle \triangleq$$
$$\lambda(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle)\ \lambda(\pi_X{:}[l_i{:}X{\to}(X{\to}\langle\!\langle B_i \rangle\!\rangle){\to}X\ ^{i\in 1..n}])\ \langle\!\langle b \rangle\!\rangle$$

$$\langle\!\langle b(A) \rangle\!\rangle \triangleq \quad \text{for } A = [l_i{:}B_i\ ^{i\in 1..n}]$$
$$\langle\!\langle b \rangle\!\rangle(\langle\!\langle A \rangle\!\rangle)\ ([l_i = \lambda(x_i{:}\langle\!\langle A \rangle\!\rangle)\ \lambda(f{:}\langle\!\langle A \rangle\!\rangle{\to}\langle\!\langle B_i \rangle\!\rangle)\ x.l_i \Leftarrow \varsigma(z{:}\langle\!\langle A \rangle\!\rangle)f(z)\ ^{i\in 1..n}])$$

$$\langle\!\langle b(Y) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle(Y)(\pi_Y)$$

Conclusions

- Structural rules for polymorphic update are sound for operational semantics. They work equally well for functional and imperative semantics.
- Structural rules can be translated into non structural rules. I have shown a translation for a restricted form of quantification.
- Theories of equality for systems with structural rules have not been studied directly yet. Similarly, theories of equality induced by the translation have not been studied.