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:

\[
\text{type Point} = [x: \text{Int}, y: \text{Int}] \\
\text{value moveXo} = \lambda(p: \text{Point}, dx: \text{Int}) \ p.x := p.x + dx; p \\
\text{value moveX} = \lambda(P <: \text{Point}) \lambda(p: P, dx: \text{Int}) \ p.x := p.x + dx; p
\]

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:

\[
\text{type Tile} = [x: \text{Int}, y: \text{Int}, hor: \text{Int}, ver: \text{Int}] \\
\text{we may want to use moveX to move tiles, not just points.}"

\[
\text{Tile} <: \text{Point} \\
\text{moveXo}([x=0, y=0, hor=1, ver=1], 1).\text{hor} \quad \text{fails} \\
\text{moveX(Tile)([x=0, y=0, hor=1, ver=1], 1).hor} \quad \text{succeeds}
\]

- 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:

\[
\text{bump} \quad \triangleq \quad \lambda(P <: \text{Point}) \lambda(p: P) \ p.x := p.x + 1 \\
\text{bump} \quad : \quad \forall(P <: \text{Point}) P \rightarrow P
\]

But ...
There is no bump there!

Neither semantically

In standard models, the type $\forall(P <: \text{Point})P \rightarrow P$ contains only the identity function.
Consider $[p]$ for any $p \in \text{Point}$. If $f : \forall(P <: \text{Point})P \rightarrow P$, then $f([p]) : [p] \rightarrow [p]$, therefore $f$ must map every point to itself, and must be the identity.

Nor parametrically

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

Nor by standard typing rules

As shown next ...

The simple rule for update

(Val Simple Update)

$$
\begin{align*}
E \vdash a : [l_i : B_i] & \quad E \vdash b : B_j \\
E \vdash a.l_j = b & : [l_i : B_i]
\end{align*}
$$

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

$$
\begin{align*}
\text{bump} & \equiv \lambda(p <: \text{Point}) \lambda(x : P) p.x := p.x + 1 \\
\text{bump} : \forall(P <: \text{Point}) P \rightarrow \text{Point}
\end{align*}
$$

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

$$
\text{bump} : \forall(P <: \text{Point}) P \rightarrow \text{Point}
$$

The “structural” rule for update

(Val Structural Update)

$$
\begin{align*}
E \vdash a : A & \quad E \vdash A <: [l_i : B_i] \quad E \vdash b : B_j \\
E \vdash a.l_j = b & : A
\end{align*}
$$

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

$$
\begin{align*}
\text{bump} & \equiv \lambda(P <: \text{Point}) \lambda(x : P) p.x := p.x + 1 \\
\text{bump} : \forall(P <: \text{Point}) P \rightarrow P
\end{align*}
$$

- Therefore, (Val Structural Update) is not sound in most semantic models, because it populates the type $\forall(P <: \text{Point})P \rightarrow P$ 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:

$$
\begin{align*}
\text{BoundedPoint} & \equiv [x : 0..9, y : 0..9] \\
\text{BoundedPoint} & <: \text{Point}
\end{align*}
$$

then:

$$
\begin{align*}
\text{bump(BoundedPoint)}(x=9, y=9) & : \text{BoundedPoint}
\end{align*}
$$

unsound!

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

$$
\begin{align*}
E \vdash B_i & \quad \forall i \in 1..m \\
E \vdash [l_i : B_i] & <: [l_i : B_i]
\end{align*}
$$

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

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Sub Object)</strong></td>
<td>( E \vdash B_i ) ( \forall i \in 1..n )</td>
<td>( E \vdash [i; B_i; \alpha_1..\alpha_m] \vdash \vdash [i; B_i; \alpha_1..\alpha_m] )</td>
</tr>
<tr>
<td><strong>(Val Subsumption)</strong></td>
<td>( E \vdash a : A ) ( E \vdash A \vdash B )</td>
<td>( E \vdash a : B )</td>
</tr>
<tr>
<td><strong>(Val Structural Update)</strong></td>
<td>( E \vdash B_i : B_i ) ( \forall i \in 1..n )</td>
<td>( E \vdash [i; B_i; \alpha_1..\alpha_m] \vdash [i; B_i; \alpha_1..\alpha_m] )</td>
</tr>
<tr>
<td><strong>(Red Update)</strong></td>
<td>( \vdash a \Rightarrow [i = v_i; \alpha_1..\alpha_m] ) ( \vdash b \Rightarrow v ) ( j \in 1..n )</td>
<td>( \vdash a, J_i := b \Rightarrow [i = v_i, J_i = v_i; \alpha_1..\alpha_m \ominus b] )</td>
</tr>
</tbody>
</table>

### The structural subtyping lemmas

**Lemma (Structural subtyping)**

If \( E \vdash [i; B_i; \alpha_i] \vdash C \) then either \( C \equiv \top \), or \( C \equiv [i; B_i; \alpha_i] \) with \( f \subseteq [i; B_i; \alpha_i] \).

If \( E \vdash C \vdash [i; B_i; \alpha_i] \) then either \( C \equiv [i; B_i; \alpha_i] \) with \( f \subseteq [i; B_i; \alpha_i] \),

or \( C \equiv X_1 \) and \( E \) contains a chain \( X_1 \vdash ... \vdash X_n \vdash [i; B_i; \alpha_i] \) with \( f \subseteq [i; B_i; \alpha_i] \).

**Proof**

By induction on the derivations of \( E \vdash [i; B_i; \alpha_i] \vdash C \) and \( E \vdash C \vdash [i; B_i; \alpha_i] \).

### Soundness by subject reduction

**Theorem (Subject reduction)**

If \( \triangleright a : A \) and \( \vdash a \Rightarrow v \) then \( \triangleright v : A \).

**Proof**

By induction on the derivation of \( \vdash a \Rightarrow v \).

**Case (Red Update)**

\[
\vdash c \Rightarrow [i = z_i; \alpha_1..\alpha_m] \quad \vdash b \Rightarrow v \quad j \in 1..n
\]

\[
\vdash a, J_i := b \Rightarrow [i = v_i, J_i = v_i; \alpha_1..\alpha_m \ominus b]
\]

By hypothesis \( \triangleright c \vdash C : A \). This must have come from (1) an application of (Val Structural Update) with assumptions \( \triangleright c \vdash C \) and \( \triangleright c \vdash C \vdash D \) where \( D \equiv [i; B_i; \alpha_1..\alpha_m] \),

and \( \triangleright b : B_i \) with conclusion \( \triangleright c, J_i := b : C \)

followed by (2) a number of subsumption steps implying \( \triangleright C \vdash A \) by transitivity.

By induction hypothesis, since \( \triangleright c \vdash C \) and \( \vdash c \Rightarrow z \equiv [i = z_i; \alpha_1..\alpha_m] \), we have \( \triangleright z : C \).

By induction hypothesis, since \( \triangleright b : B_i \) and \( \vdash b \Rightarrow v \), we have \( \triangleright v : B_i \).

Now, \( \triangleright z : C \) must have come from (1) an application of (Val Object) with assumptions \( \triangleright z_i : B_i \) and \( C' \equiv [i; B_i; \alpha_1..\alpha_m] \),

and with conclusion \( \triangleright z : C' \), followed by (2) a number of subsumption steps implying \( \triangleright C' \vdash C \) by transitivity. By transitivity,

\( \triangleright C' \vdash C \vdash D \).

Hence by the Structural Subtyping Lemma, we must have \( B_i \equiv B_i' \). Thus \( \triangleright v : B_i' \).

Then, by (Val Object), we obtain \( \triangleright [i = w_i, J_i = z_i; \alpha_1..\alpha_m \ominus b] : C' \).

Since \( \triangleright C' : A \) by transitivity, we have \( \triangleright [i = w_i, J_i = z_i; \alpha_1..\alpha_m \ominus b] : A \) 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 \( (a, b) \) and \( (c, d) \) 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 \( C_1 \times C_2 \).

\[
\frac{E \vdash C : C_1 \times C_2 \quad E \vdash a : C_1 \quad E \vdash b : C_2}{E \vdash (a, b) : C}
\]

The soundness of this rule depends on the property that every subtype of a product type \( C_1 \times C_2 \) 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 \( C_1 \times C_2 \) whenever \( a \) and \( b \) are elements of \( C_1 \times C_2 \). If \( a \) and \( b \) are different, then \( (\text{fst}(a), \text{snd}(b)) \) is not an element of \( \{a, b\} \). Note that \( \{a, b\} \) is not a product type.

A structural rule for method invocation

M. Abadi

- In the context of object types with Self types:

\[
\frac{\text{Val Select}}{E \vdash a : A \quad E \vdash A <\text{Obj}(X)[l_i:B_i(X)] \quad j \in 1..n}{E \vdash a_j : B_i[A]}
\]

This structural rule is necessary to “encapsulate” structural update inside methods:

\[
A \triangleq \text{Obj}(X)[n : \text{Int}, \text{bump} : X] \\
\lambda (Y < A) \lambda (y : Y) \text{y.bump} \\
\text{: } \forall (Y < A) Y \rightarrow Y
\]

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:

\[
[i : B_i(X)]^{\leq 1..n} \triangleq \mu(Y) \exists (X < Y) (r : X, l_i^{\text{rel}}, X \rightarrow B_i(X))^{\leq 1..n}, l_i^{\text{rel}}(X \rightarrow B_i(X) \rightarrow X)^{\leq 1..n}
\]

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:

\[
\frac{E \vdash C \triangleq \text{Obj}(X)[B(X)] \quad E \vdash a : C}{E \vdash \text{unfold}(a) : B[C]}
\]

Structural rules and class encodings

Types of the form \( \forall (X < A) X \rightarrow B(X) \) 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.

\[
A \triangleq \text{Obj}(X)[l_i : B_i(X)]^{\leq 1..n}
\]

\[
\text{Class}(A) \triangleq [\text{new} : A, l_i : \forall (X < A) X \rightarrow B_i(X)]^{\leq 1..n}
\]

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

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

\[
c : \text{Class}(\text{Bump}) \triangleq
\begin{align*}
\{\text{new} = \xi(c : \text{Class}(\text{Bump})) \mid n = 0, & \text{bump} = \xi(s : \text{Bump}) c.\text{bump}(\text{Bump})(s), \\
& \text{bump} = \lambda (X < \text{Bump}) \lambda (x : X) x.n = x.n + 1\}
\end{align*}
\]
A structural rule for cloning

- In the context of imperative object calculi:

\[
\begin{align*}
(Vl\ Clone) & \\
E \vdash a : A & \quad E \vdash A <: [l; B_l; ^{i+1..n}] \\
& \quad j \in 1..n \\
E \vdash \text{clone}(a) : A
\end{align*}
\]

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

\[
\begin{align*}
\text{bump} & \triangleq \lambda (P <: \text{Point}) \lambda (p : P) \text{clone}(p).x := p.x + 1 \\
\text{bump} : \forall (P <: \text{Point}) P \rightarrow P
\end{align*}
\]

Translating away structural rules

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

\[
\begin{align*}
f : \forall (X <: [l: \text{Int}]) X \rightarrow X & \triangleq \\
\lambda (X <: [l : \text{Int}]) \lambda (x : X) x.l := 3 \\
f ([l: \text{Int}]) & \quad \text{(N.B. the update } x.l := 3 \text{ uses the structural rule)}
\end{align*}
\]

translates to:

\[
\begin{align*}
f : \forall (X <: [l: \text{Int}]) [l : \text{Int} \rightarrow X] \rightarrow X & \triangleq \\
\lambda (X <: [l : \text{Int}]) \lambda (\pi x : [l : \text{Int} \rightarrow X]) \lambda (x : X) \pi x . l(x)(3) \\
f ([l : \text{Int}]) ([l = \lambda (x : [l : \text{Int}]) \lambda (y : \text{Int}) x.l := y]) & \quad \text{(N.B. the update } x.l := y \text{ uses the non-structural rule)}
\end{align*}
\]

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

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 semantics, 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 $\forall (X <: \text{Point}) X \rightarrow X$ 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.

Syntax

\[
\begin{align*}
A, B & ::= \\
X & \quad \text{type variable} \\
[l; B_l; ^{i+1..n}] & \quad \text{object type } (l; \text{distinct}) \\
A \rightarrow B & \quad \text{function types} \\
\forall (X <: [l; B_l; ^{i+1..n}])B & \quad \text{bounded universal type} \\
\pi a : \Gamma & ::= \\
\pi x : A & \quad \text{variable} \\
\pi i & \quad \text{object } (l; \text{distinct}) \\
\pi a . \pi x : A & \quad \text{method invocation} \\
\pi f : \pi a & \quad \text{method update} \\
\pi (f ; a) & \quad \text{function} \\
\pi a & \quad \text{application} \\
\pi (\lambda (X : [l; B_l; ^{i+1..n}])b) & \quad \text{polymorphic function} \\
\pi b (A) & \quad \text{polymorphic instantiation}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Env $\emptyset$)</td>
<td>$E \vdash A \land x \notin \text{dom}(E)$</td>
</tr>
<tr>
<td>(Env $x$)</td>
<td>$E \vdash A \land x \notin \text{dom}(E)$</td>
</tr>
<tr>
<td>(Env $X &lt; x$)</td>
<td>$E \vdash A \land X \subseteq \text{dom}(E)$</td>
</tr>
</tbody>
</table>

Typing

- (Val Subsumption): $E \vdash a : A \vdash x : A \land x \notin \text{dom}(E)$
- (Val x): $E \vdash A < : B$
- (Val Select): $E \vdash a : [i : B_i^{n+m}]$
- (Val Object): $E \vdash [i : X_i] : [i : B_i^{n+m}]$
- (Val Update Obj) (where $A = [i : B_i^{n+m}]$): $E \vdash a : A \vdash x : A \land x \notin \text{dom}(E)$
- (Val Fun): $E \vdash \lambda(x : A)b : A \rightarrow B$
- (Val Fun2$\rightarrow$): $E \vdash \lambda(x : A)b : A \rightarrow B$
- (Val Appl): $E \vdash b : A \rightarrow B \vdash a : A$
- (Val Appl2$\rightarrow$): $E \vdash b : A \rightarrow B \vdash a : [i : B_i^{n+m}]$

Subtyping

- (Sub Refl): $E \vdash A \vdash A < : B$
- (Sub Trans): $E \vdash A < : B \vdash B < : C$
- (Sub X): $E \vdash X \subseteq \text{dom}(E)$
- (Sub Object): $E \vdash [i : B_i^{n+m}]$
- (Sub Arrow): $E \vdash A < : B \vdash A' < : B'$
- (Sub All): $E \vdash \forall (e : X) B < : \forall (e : X') B'$

- 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

- ($\emptyset$) $\equiv \emptyset$
- $(E, x : A) \equiv \langle E \rangle, x : \langle A \rangle$
- $(E, X : [i : B_i^{n+m}]) \equiv \langle E \rangle, X : [i : B_i^{n+m}], \pi_X([i : X \rightarrow (X \rightarrow B_i)] \rightarrow X^{n+m})$

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

- 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

- ($\emptyset$) $\equiv \emptyset$
- $(E, x : A) \equiv \langle E \rangle, x : \langle A \rangle$
- $(E, X : [i : B_i^{n+m}]) \equiv \langle E \rangle, X : [i : B_i^{n+m}], \pi_X([i : X \rightarrow (X \rightarrow B_i)] \rightarrow X^{n+m})$

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

- 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

- ($\emptyset$) $\equiv \emptyset$
- $(E, x : A) \equiv \langle E \rangle, x : \langle A \rangle$
- $(E, X : [i : B_i^{n+m}]) \equiv \langle E \rangle, X : [i : B_i^{n+m}], \pi_X([i : X \rightarrow (X \rightarrow B_i)] \rightarrow X^{n+m})$

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

- 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 Types

\[
\begin{align*}
\llbracket X \rrbracket & \triangleq X \\
\llbracket [i: B_i]^{i \in L} \rrbracket & \triangleq [i: \llbracket B_i \rrbracket^{i \in L}] \\
\llbracket A \rightarrow B \rrbracket & \triangleq \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \\
\forall (X: [i: B_i]^{i \in L}) B & \triangleq \forall (X: \llbracket [i: B_i]^{i \in L} \rrbracket) [i: X \rightarrow (X \rightarrow \llbracket B \rrbracket) \rightarrow \llbracket B \rrbracket]
\end{align*}
\]

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

\[
\forall (X: [i: B_i]^{i \in L}) B \lll \forall (X: \llbracket [i: B_i]^{i \in L} \rrbracket) B
\]

since:

\[
\forall (X: \llbracket [i: B_i]^{i \in L} \rrbracket) [i: X \rightarrow (X \rightarrow \llbracket B \rrbracket) \rightarrow \llbracket B \rrbracket] \lll \\
\forall (X: [i: B_i]^{i \in L}) [i: X \rightarrow (X \rightarrow \llbracket B \rrbracket) \rightarrow \llbracket B \rrbracket]
\]

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

Translation of Terms

\[
\begin{align*}
\llbracket x \rrbracket & \triangleq x \\
\llbracket [i: (x: A_i)]^{i \in L} \rrbracket & \triangleq [i: \llbracket (x: A_i) \rrbracket^{i \in L}] \\
\llbracket [a: l] \rrbracket & \triangleq [a: l] \\
\llbracket \lambda (x: A) \rrbracket & \triangleq \lambda (x: \llbracket A \rrbracket) \\
\llbracket b(a) \rrbracket & \triangleq \llbracket b \rrbracket [\llbracket a \rrbracket] \\
\llbracket \lambda (x: A) b \rrbracket & \triangleq \lambda (x: \llbracket A \rrbracket) [\llbracket b \rrbracket] \\
\llbracket \lambda (x: [i: B_i]^{i \in L}) b \rrbracket & \triangleq \\
\lambda (x: \llbracket [i: B_i]^{i \in L} \rrbracket) [\lambda (x: \llbracket [i: [B_i]^{i \in L}] \rrbracket) [i: X \rightarrow (X \rightarrow \llbracket B \rrbracket) \rightarrow \llbracket B \rrbracket] \rightarrow \llbracket B \rrbracket] \rightarrow \llbracket B \rrbracket]
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{Val Update Obj} \rrbracket & \triangleq \text{Val Update Obj} \\
\llbracket \text{Val Update X} \rrbracket & \triangleq \text{Val Update X}
\end{align*}
\]

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.