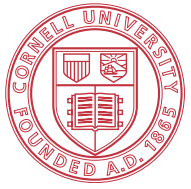
Languages for Bidirectional Transformations

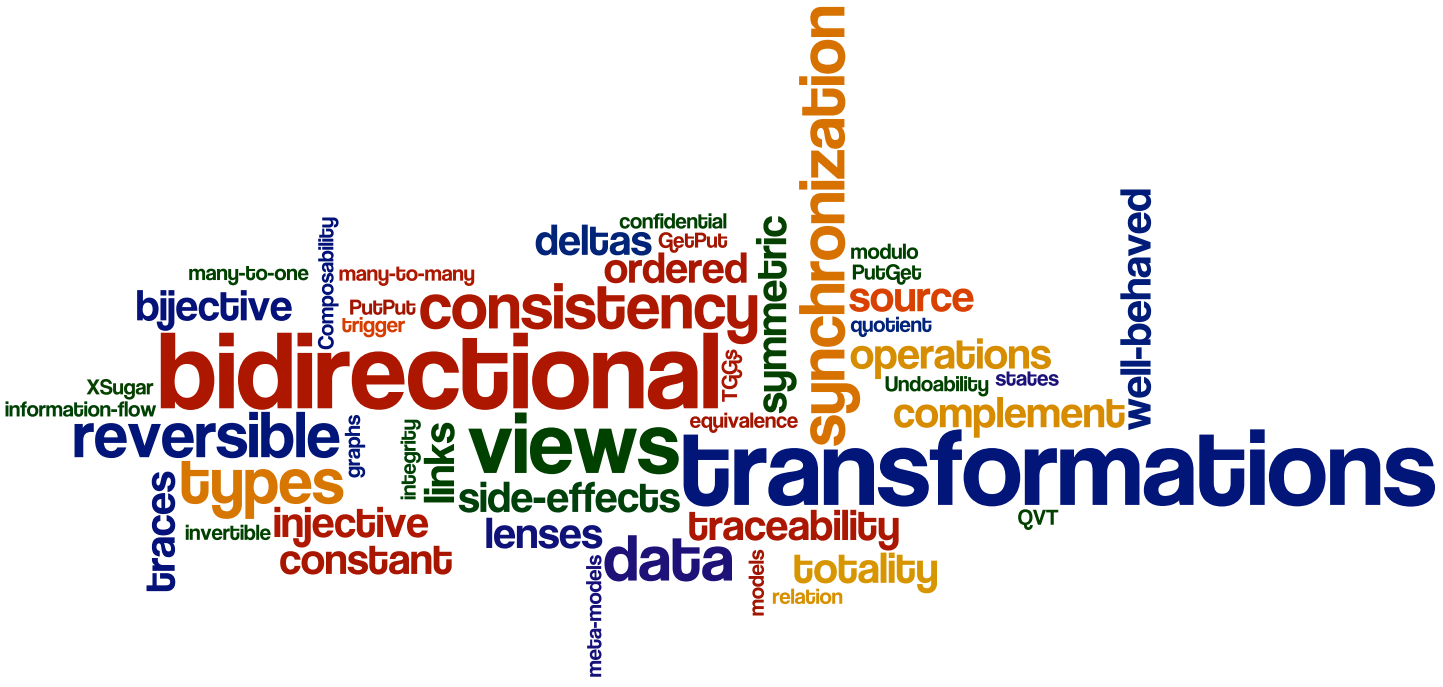
Nate Foster (Cornell)
Robert Glück (DIKU)
Martin Hofmann (LMU)
Zhenjiang Hu (NII)
Benjamin Pierce (Penn)
Janis Voigtländer (Bonn)



BX Dagstuhl Seminar

17 January 2011

Objective



Goals:

- Present the key semantic issues in a clean setting
- Study similarities and differences between languages
- Provide a common vocabulary for the meeting

Plan

Part I: Semantics

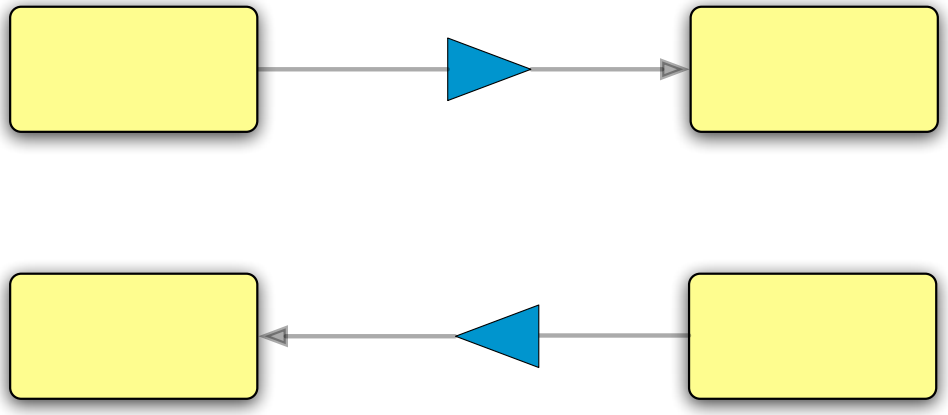
> *"The use of [QVT-style] bidirectional transformations has not spread fast, despite the early availability of a few tools, partly (we think) because of uncertainty among users over fundamental semantic issues."* *[Stevens '09]*

Part II: Mechanisms

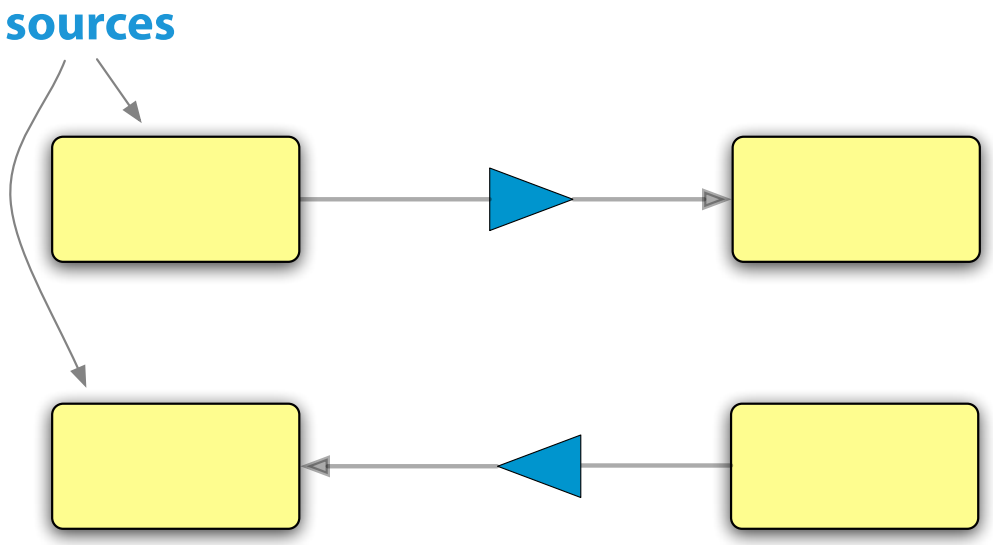
- Survey approaches used in several different languages
- Identify open questions

Goal: accessible to *everyone* $\Longrightarrow$ ask questions!

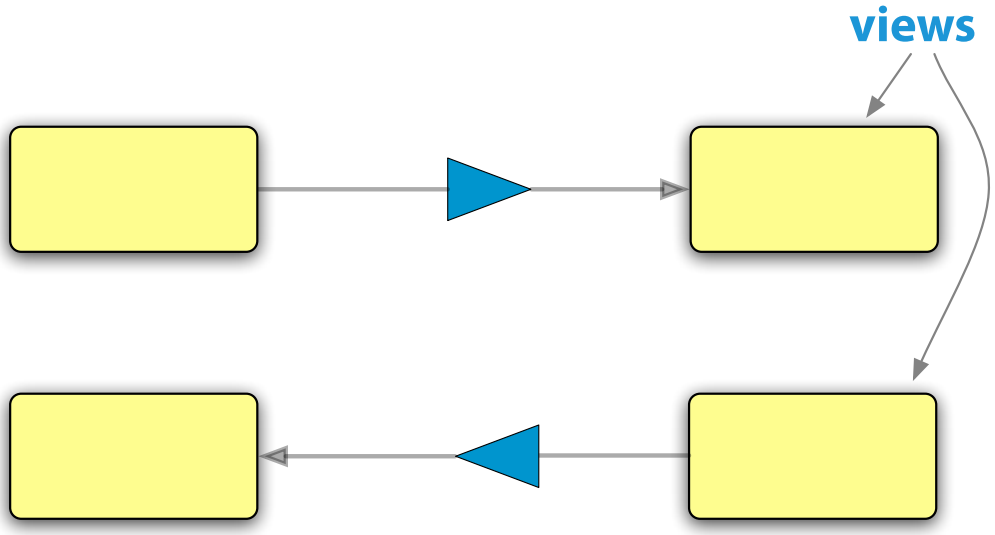
Terminology



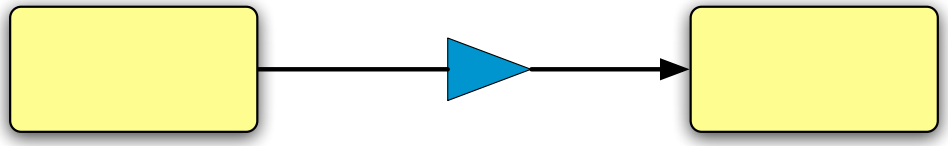
Terminology



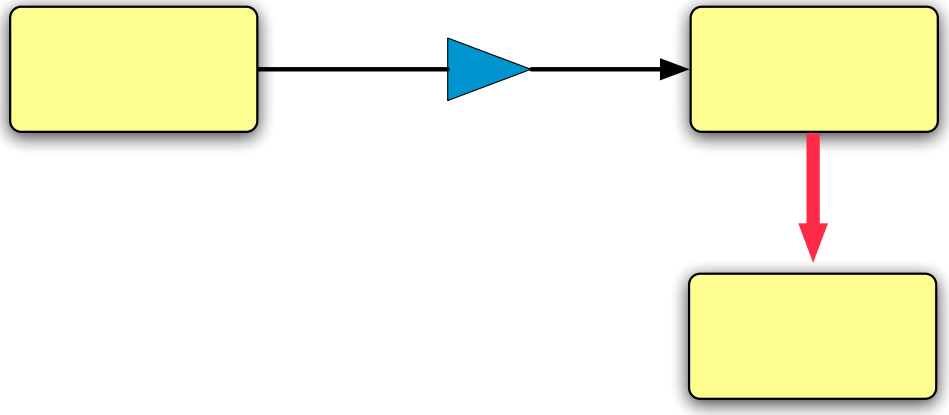
Terminology



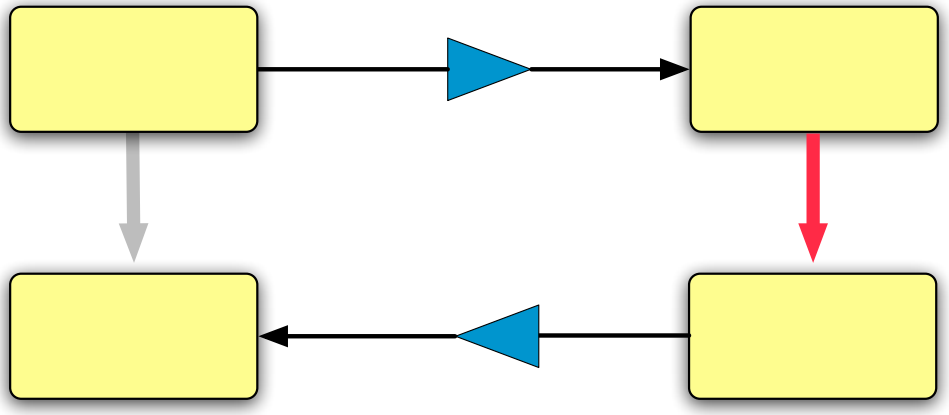
Terminology



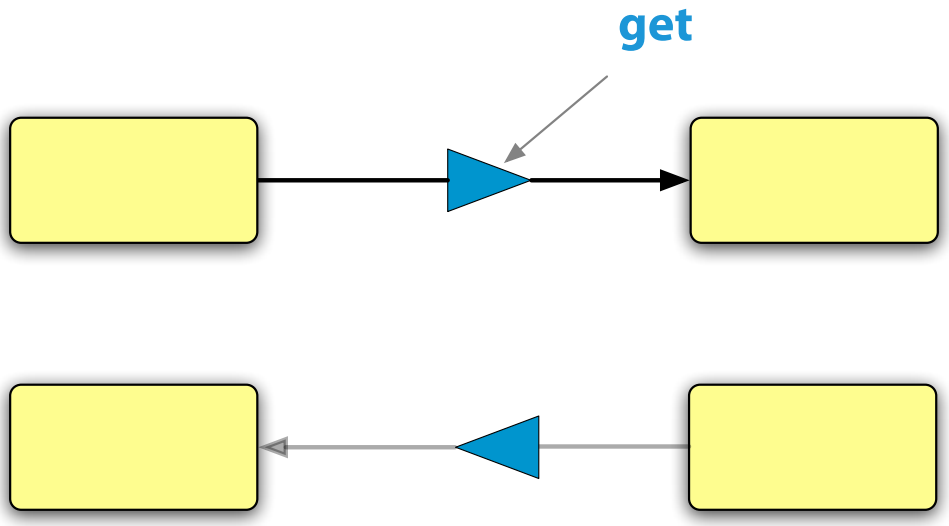
Terminology



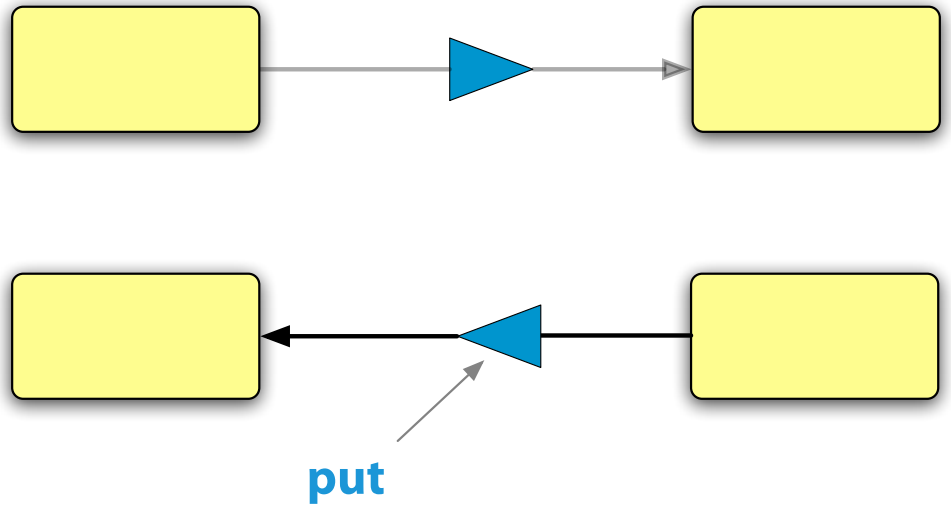
Terminology



Terminology



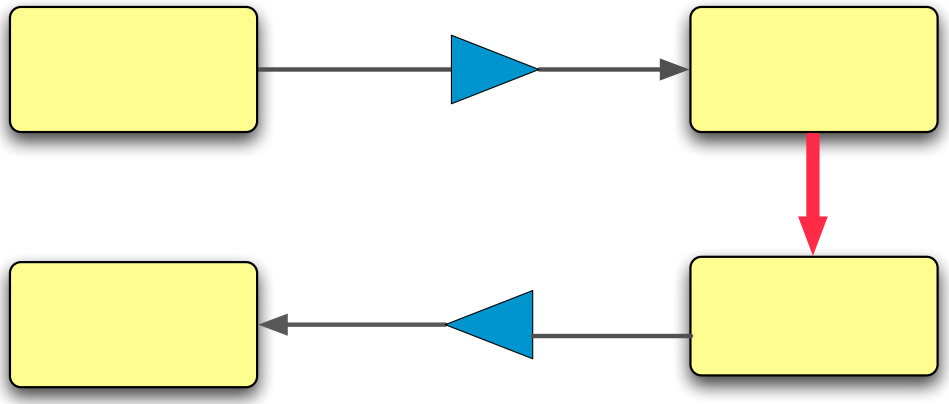
Terminology



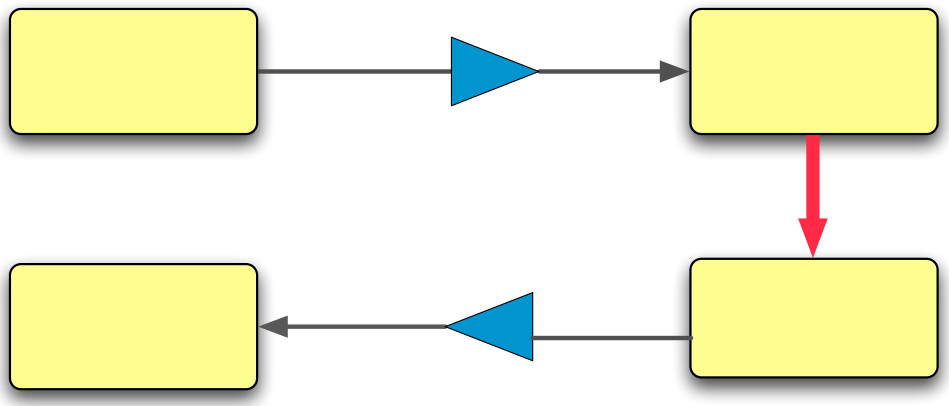
Question #1:

What do we provide to the **put** *function?*

State-based vs. operation-based

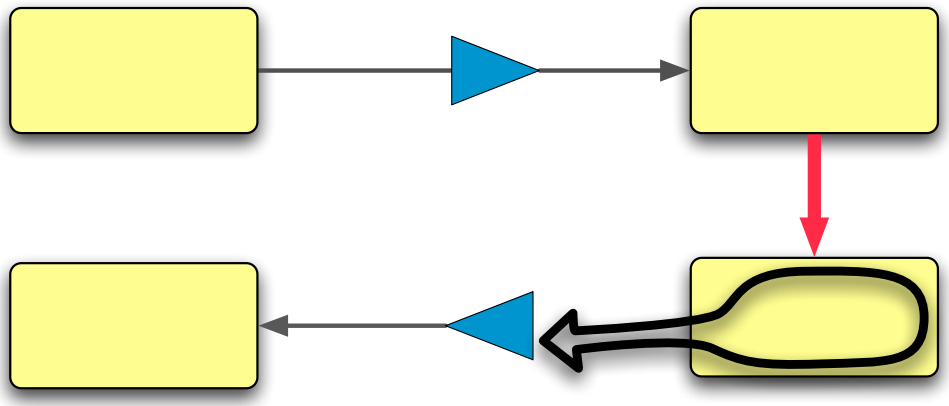


State-based vs. operation-based



Do we give it...

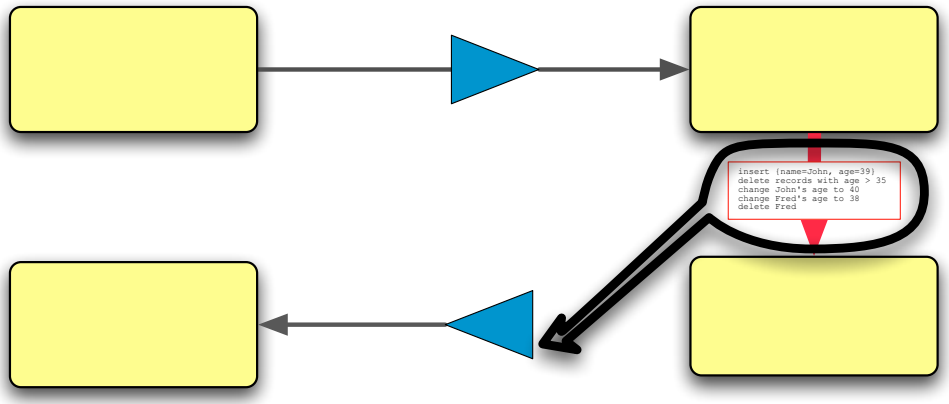
State-based vs. operation-based



Do we give it...

- the new state of the view?

State-based vs. operation-based



Do we give it...

- the new state of the view?
- (a description of) the update applied to the view?

Both of these are reasonable answers.

State-based vs. operation-based

Tradeoffs:

- State-based approach:
  - \+ mathematically simpler
  - \+ easier to build "loosely coupled" systems: **put** does not need to know what update was applied, just the result
- Operation-based approach:
  - \+ provides **put** with more information
  - \+ captures intuition of "manipulating (small) deltas to (huge) structures"

We'll focus on the simpler state-based approach.

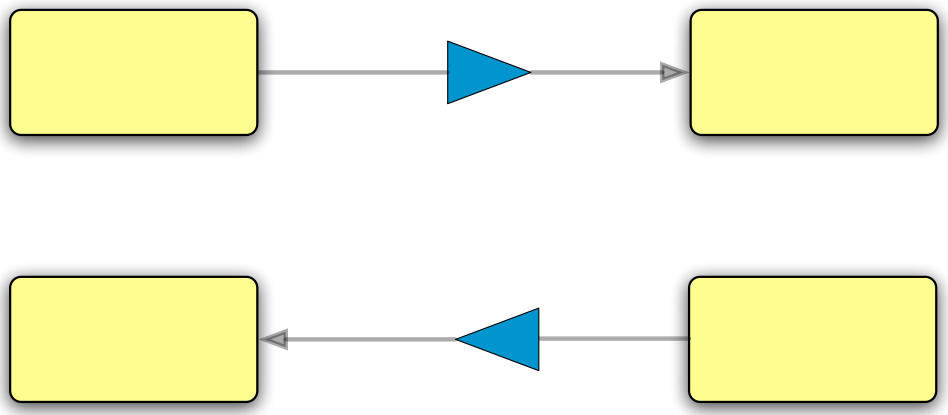
See [Diskin, Xiong, and Czarnecki '10] for more...

Question #2:

Can the **get** *function be used to hide part of the source?*

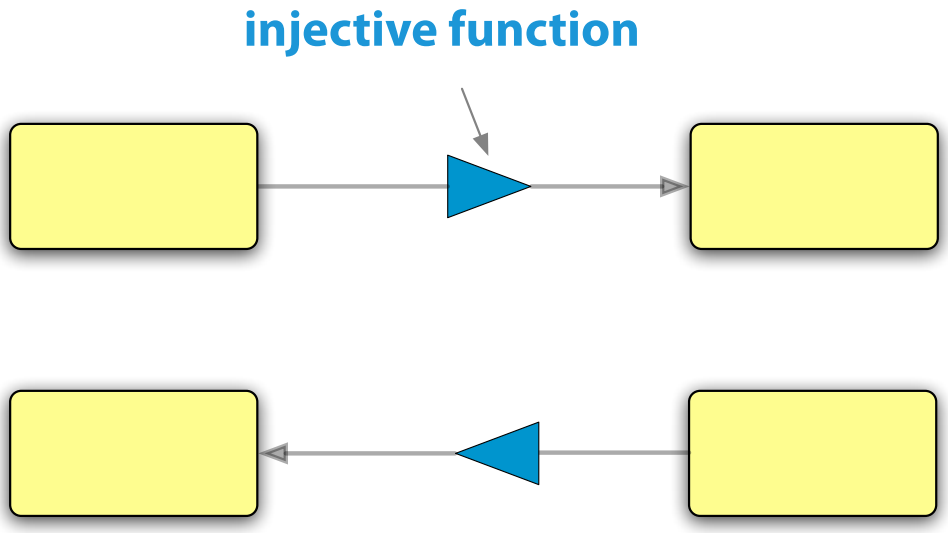
Bijective vs. Bidirectional

- Formally, can the **get** function be non-injective?



Bijective vs. Bidirectional

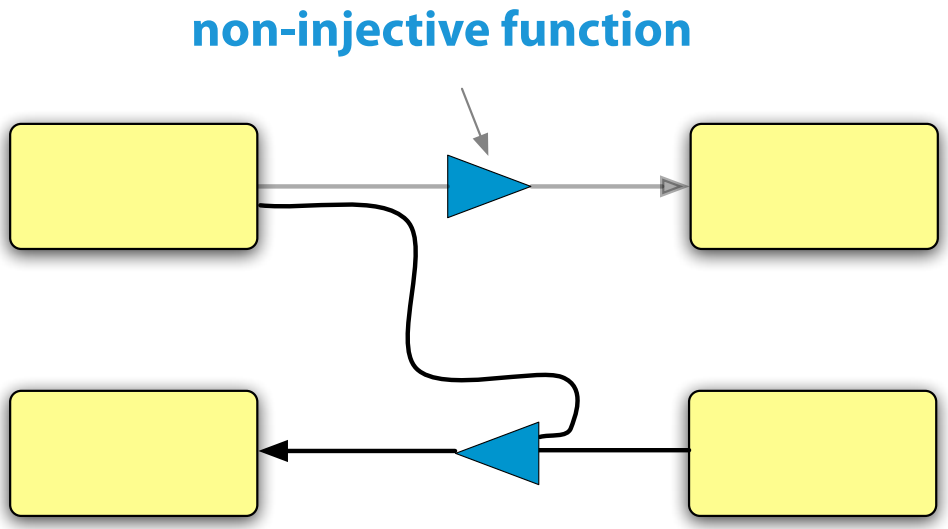
- Formally, can the **get** function be non-injective?
- No $\Longrightarrow$ **put** can map the view directly to a source.



$$\textbf{get} \in S \to V$$
$$\textbf{put} \in V \to S$$

Bijective vs. Bidirectional

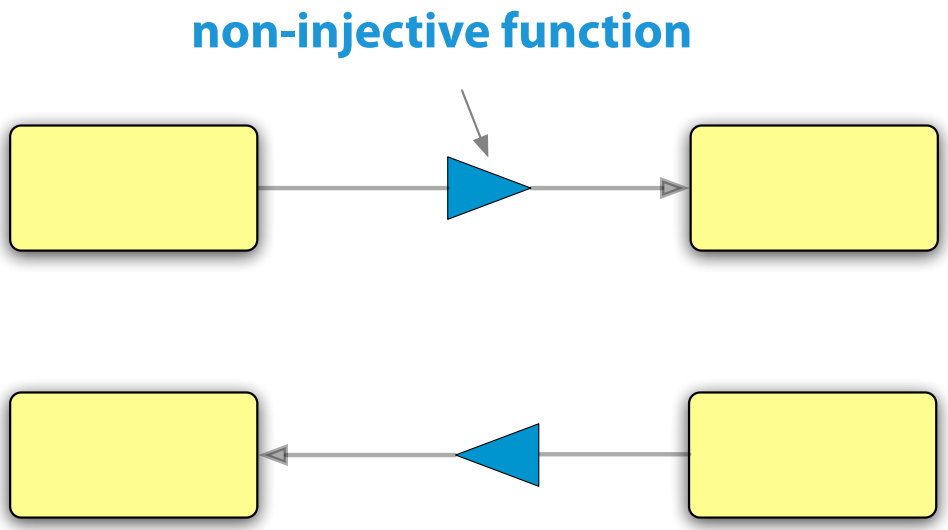
- Formally, can the **get** function be non-injective?
- No $\Longrightarrow$ **put** can map the view directly to a source.
- Yes $\Longrightarrow$ **put** needs to take the source as an input.



$$\textbf{get} \in S \to V$$
$$\textbf{put} \in V \times S \to S$$

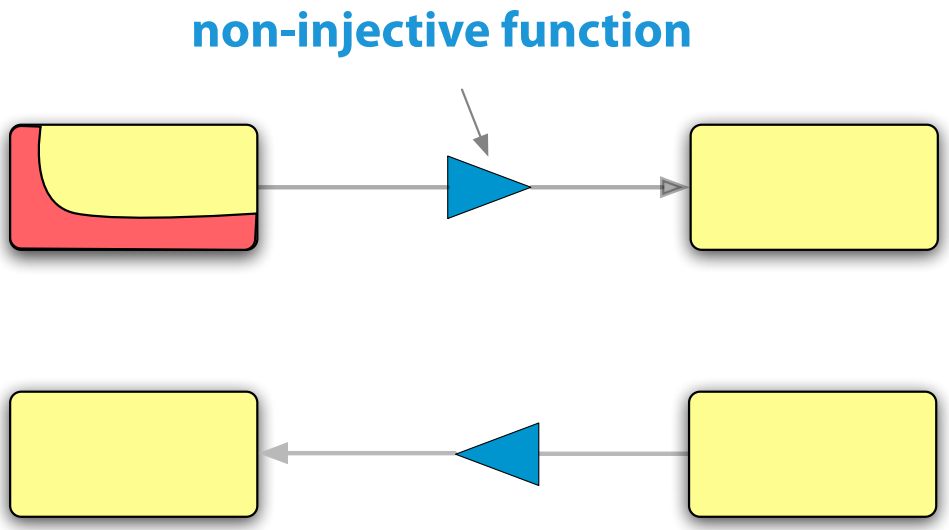
Complements

We can refine this architecture by representing the complement to the view explicitly.



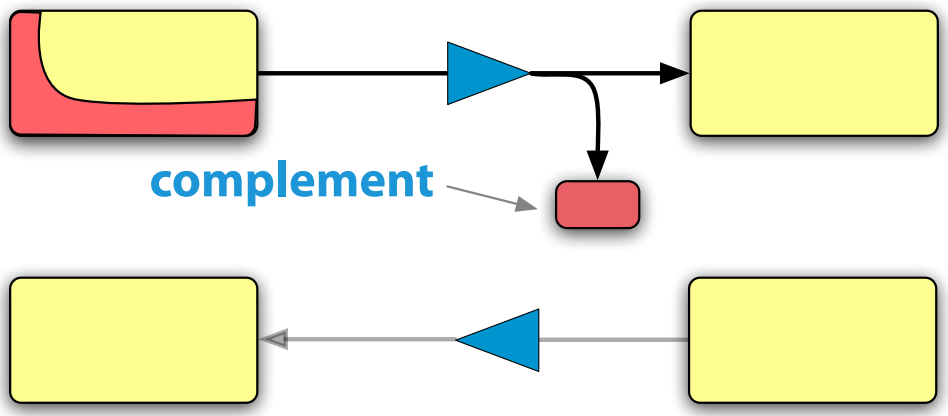
Complements

We can refine this architecture by representing the complement to the view explicitly.



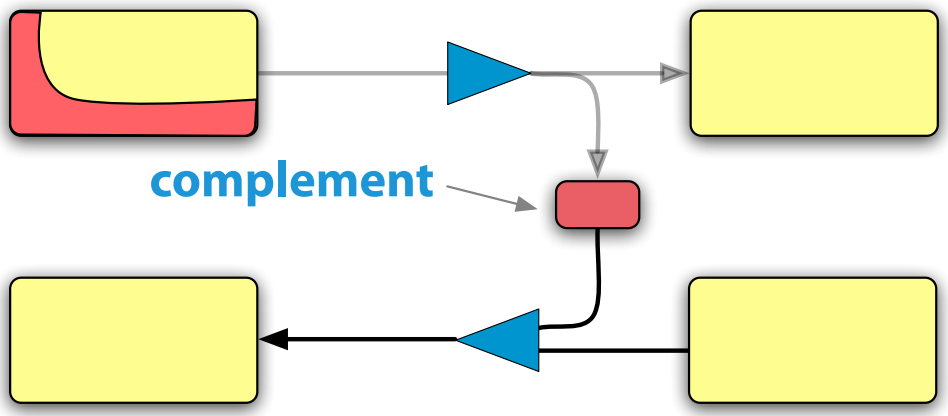
Complements

We can refine this architecture by representing the complement to the view explicitly.



Complements

We can refine this architecture by representing the complement to the view explicitly.

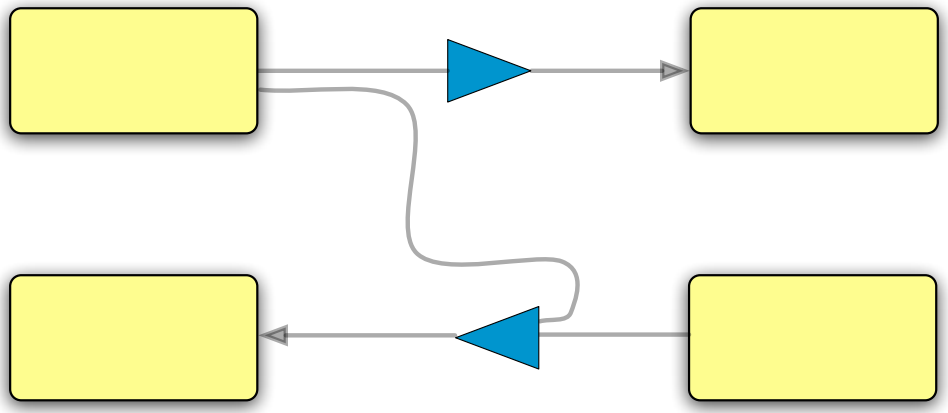


$$\textbf{get} \in S \to V \times C$$
$$\textbf{put} \in V \times C \to S$$

A Digression on Symmetry...

What about the other direction? Can the **put** function be used to hide information in the view?

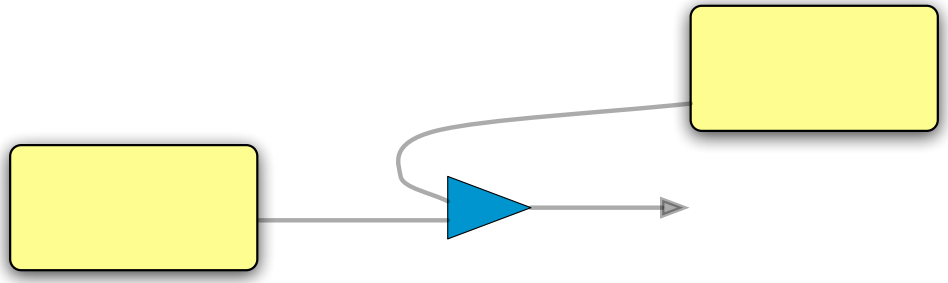
That is, can we make everything symmetric? [Stevens '09]



A Digression on Symmetry...

What about the other direction? Can the **put** function be used to hide information in the view?

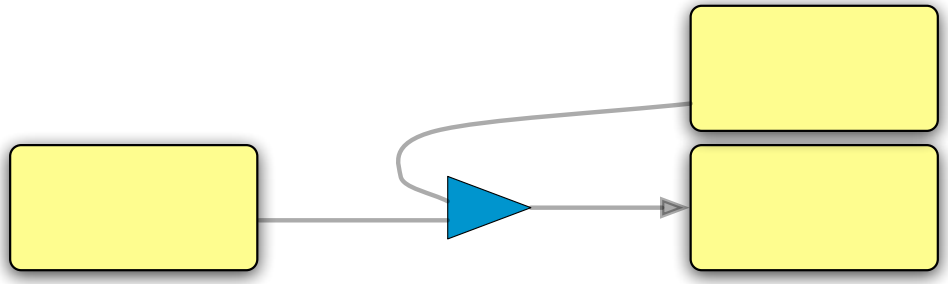
That is, can we make everything symmetric? [Stevens '09]



A Digression on Symmetry...

What about the other direction? Can the **put** function be used to hide information in the view?

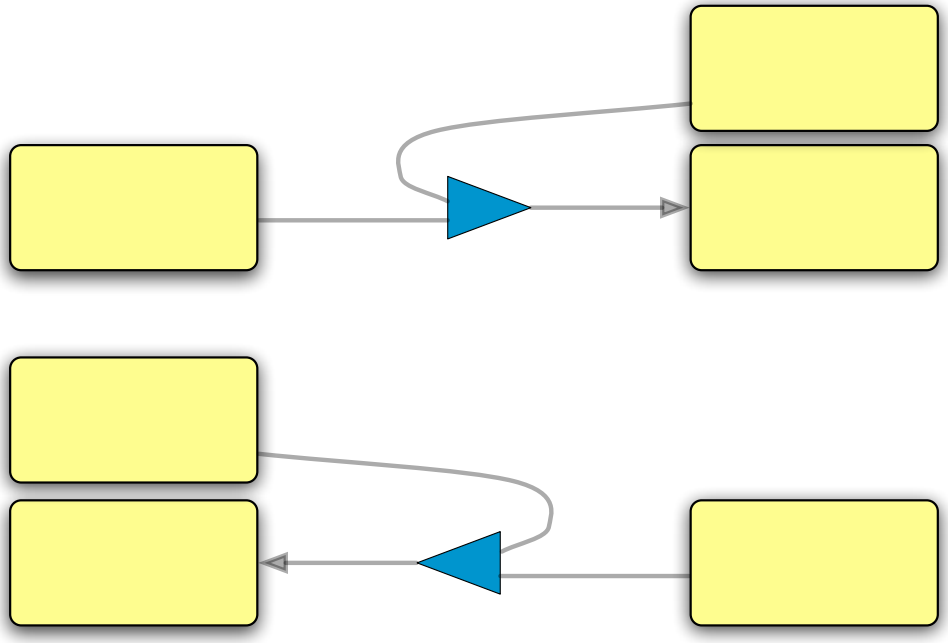
That is, can we make everything symmetric? [Stevens '09]



A Digression on Symmetry...

What about the other direction? Can the **put** function be used to hide information in the view?

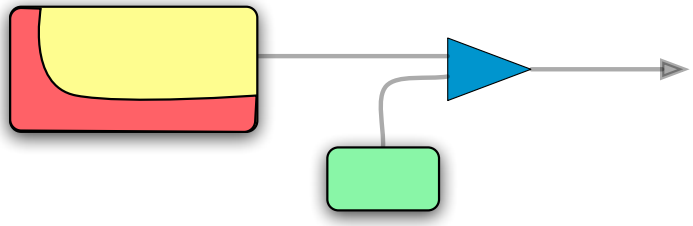
That is, can we make everything symmetric? [Stevens '09]



$$\textbf{get} \quad \in \quad S \times V \to V$$
$$\textbf{put} \quad \in \quad V \times S \to S$$

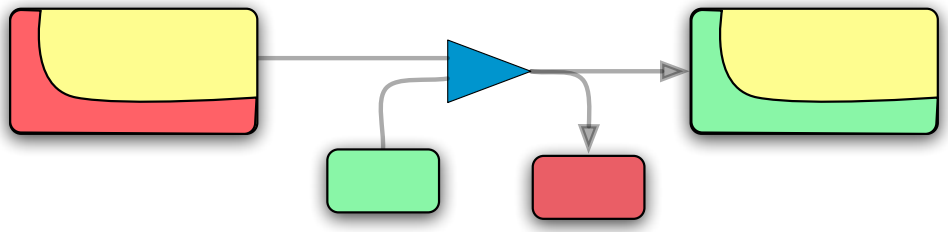
Symmetry + Complements

...and we can refine the framework again by introducing complements [Hofmann, Pierce, Wagner '11]



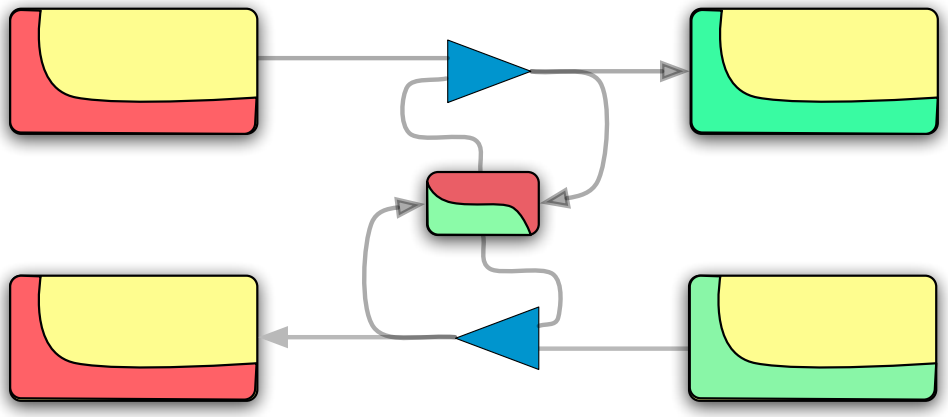
Symmetry + Complements

...and we can refine the framework again by introducing complements [Hofmann, Pierce, Wagner '11]



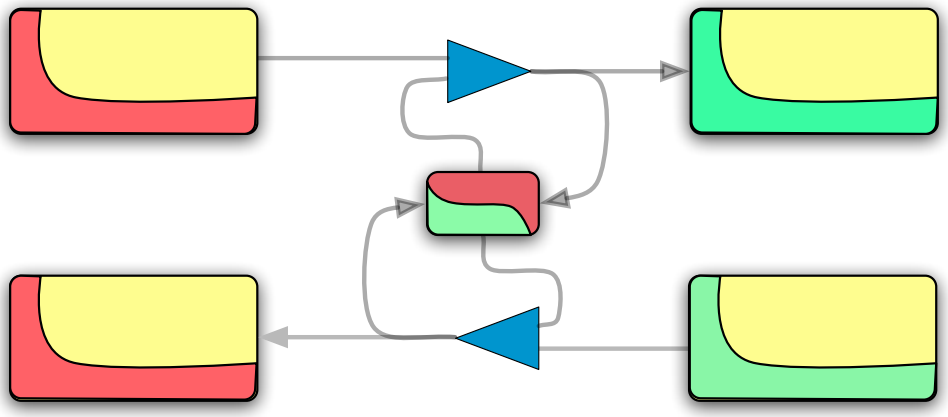
Symmetry + Complements

...and we can refine the framework again by introducing complements [Hofmann, Pierce, Wagner '11]



Symmetry + Complements

...and we can refine the framework again by introducing complements [Hofmann, Pierce, Wagner '11]



$$\textbf{get} \in S \times C \to V \times C$$
$$\textbf{put} \in V \times C \to S \times C$$

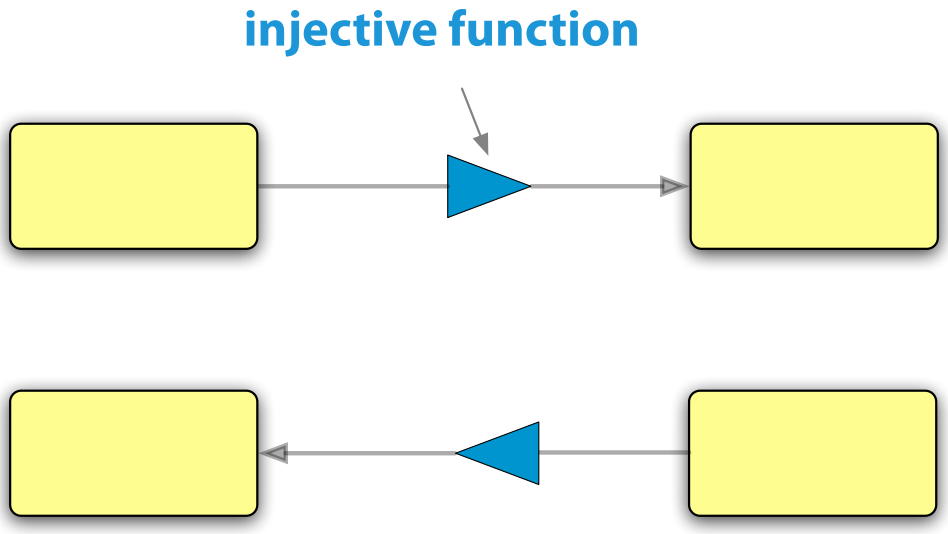
(...and these symmetric lenses compose!)

Question #3:

What constraints do we need place on **get** *and* **put** *to ensure that they work well together?*

An Easy Case

With bijective transformations...

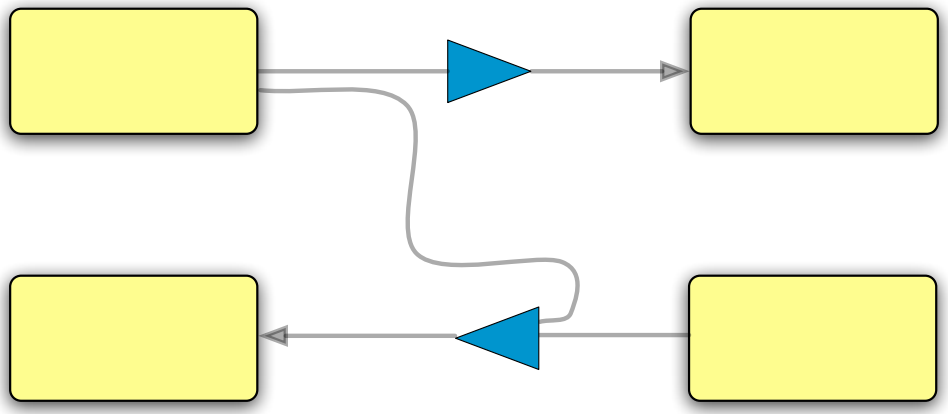


...the desired behavior is obvious

$$\textbf{put}\ (\textbf{get}\ s) = s$$
$$\textbf{get}\ (\textbf{put}\ v) = v$$

The General Case

But for bidirectional transformations...

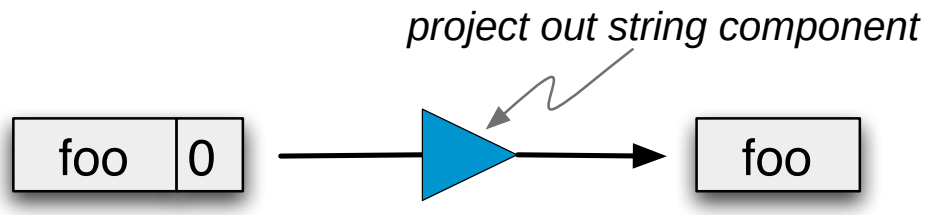


We need to identify conditions that allow us to

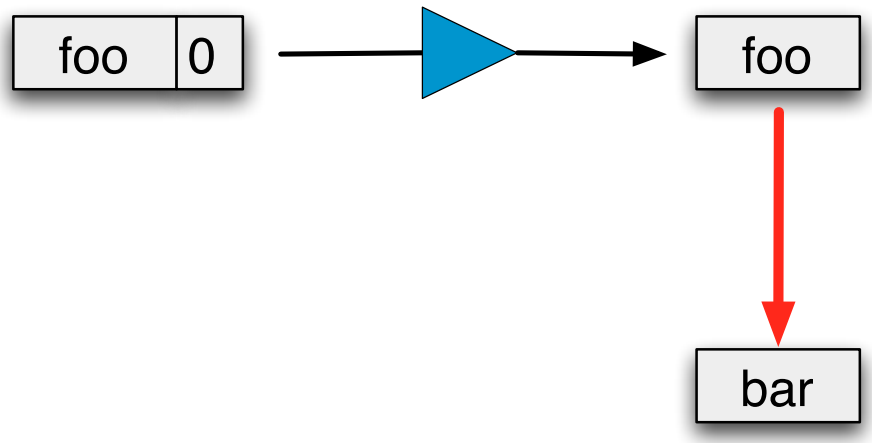
- recognize and reject bad (unreasonable) primitives
- understand and predict behavior

An Unreasonable Example

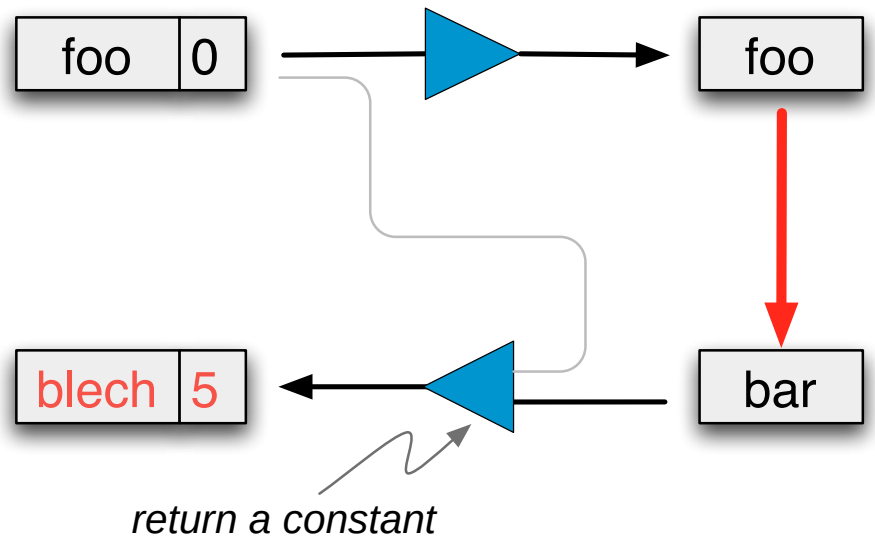
An Unreasonable Example



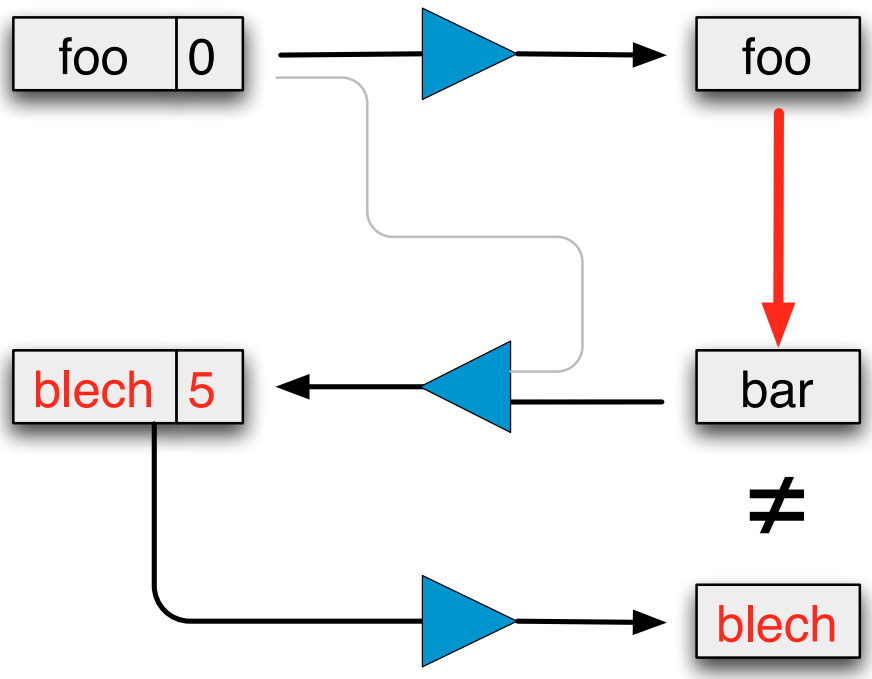
An Unreasonable Example



An Unreasonable Example



An Unreasonable Example



The PutGet law

Principle:

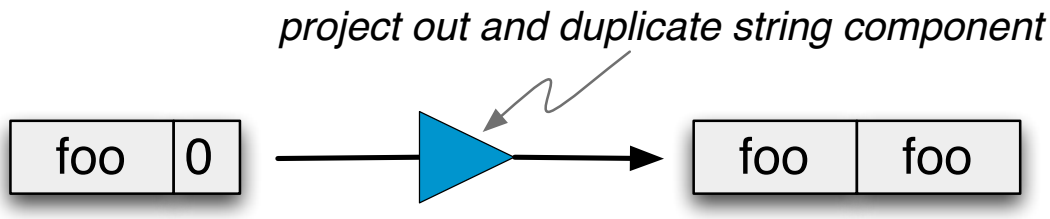
Updates should be "translated exactly" — i.e., to a source structure for which **get** *yields exactly the updated target structure.*

Formally:

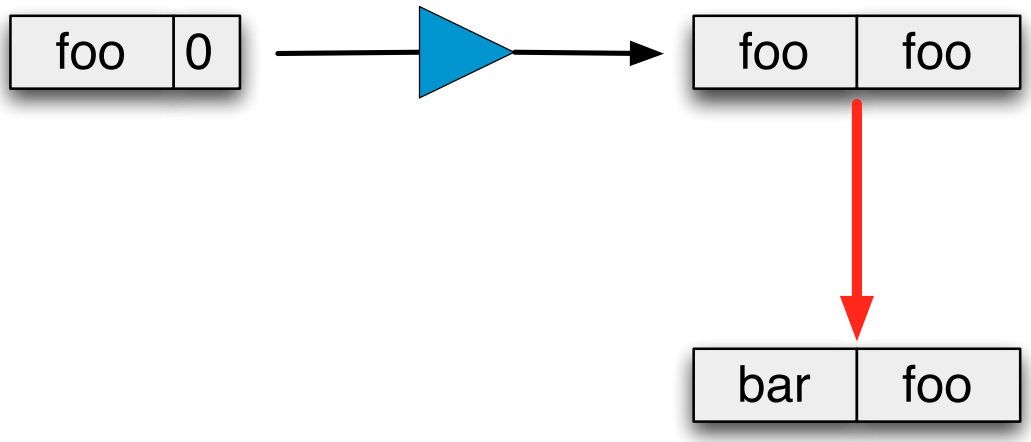
$$\textbf{get}\ (\textbf{put}\ v\ s) \quad = \quad v$$

A Debatable Example

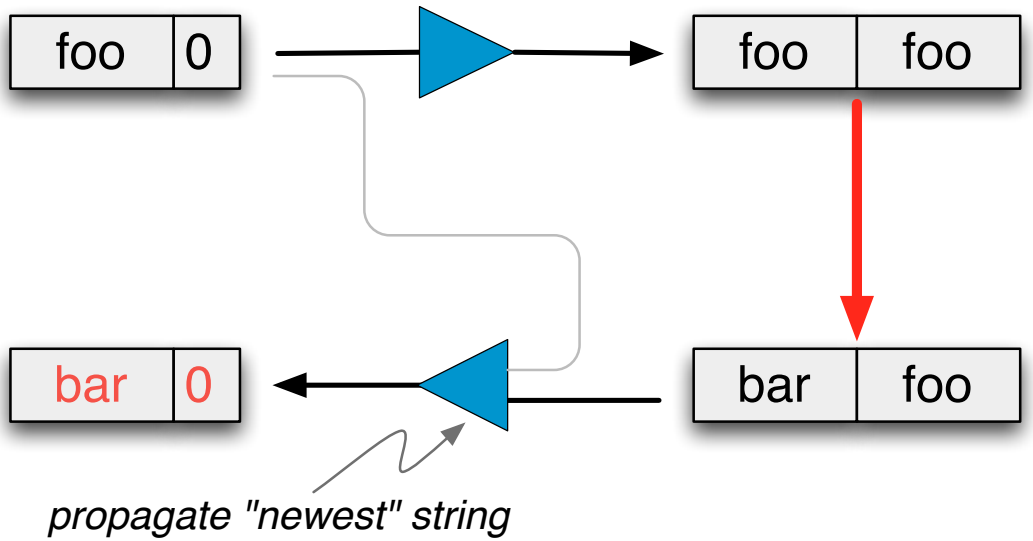
A Debatable Example



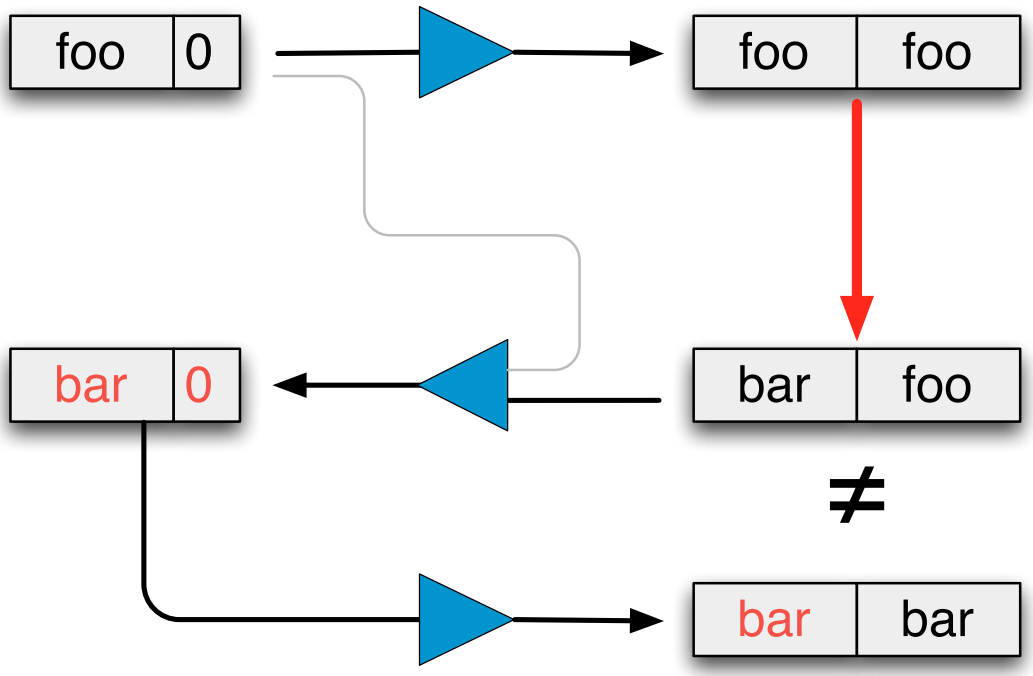
A Debatable Example



A Debatable Example



A Debatable Example



Weakening the PutGet law

If we want to allow such behavior, we need to weaken PutGet. Here is one possibility:

$$\frac{\textbf{put}\ v\ s = s' \qquad \textbf{get}\ s' = v'}{\textbf{put}\ v'\ s = s'}$$

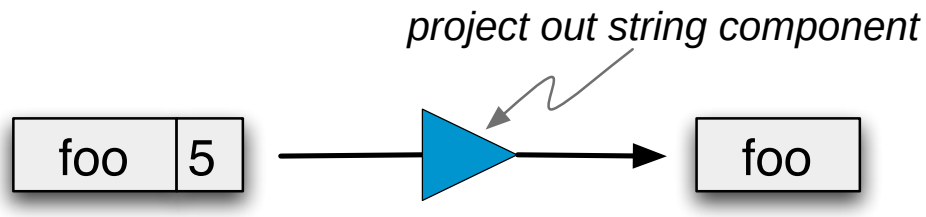
Intuition:

Propagating an update may have "side-effects", but only on the initial round-trip.

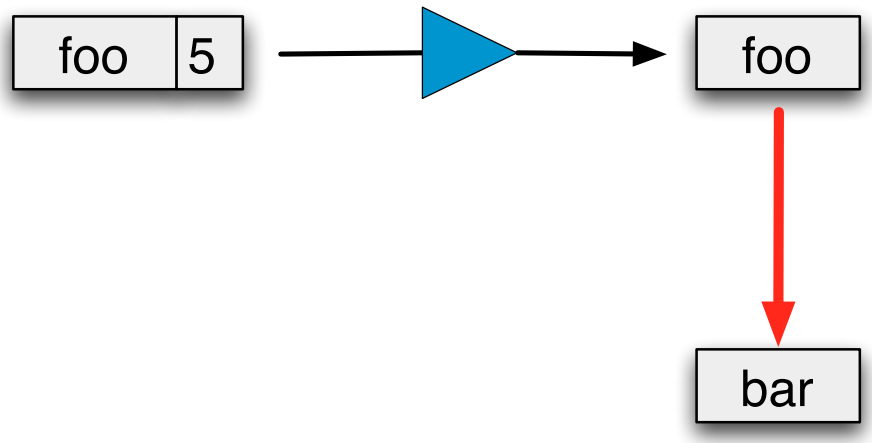
Similar idea in databases:

Propagating an update must have "minimal side-effects" on the view.

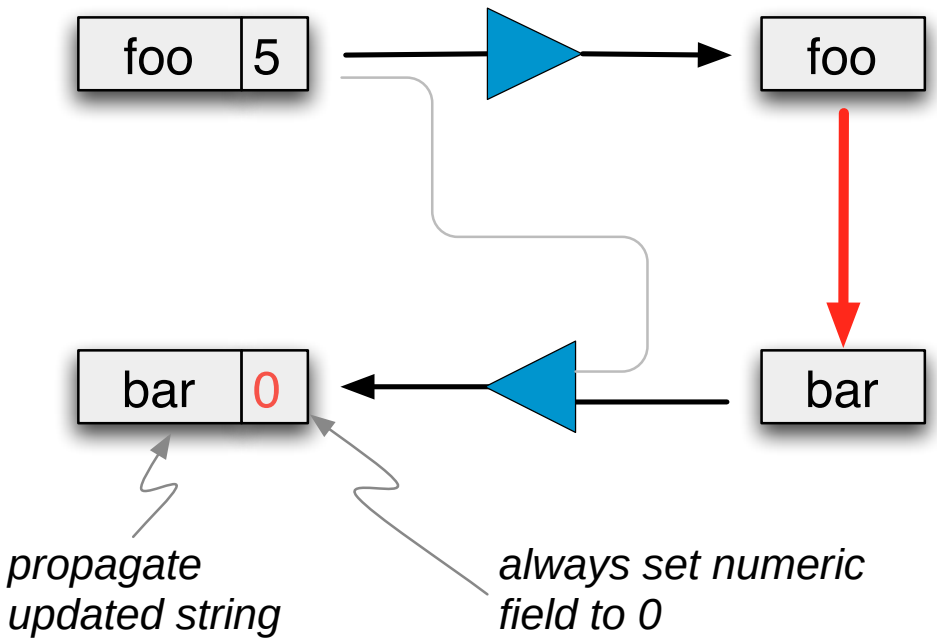
Another Unreasonable Example



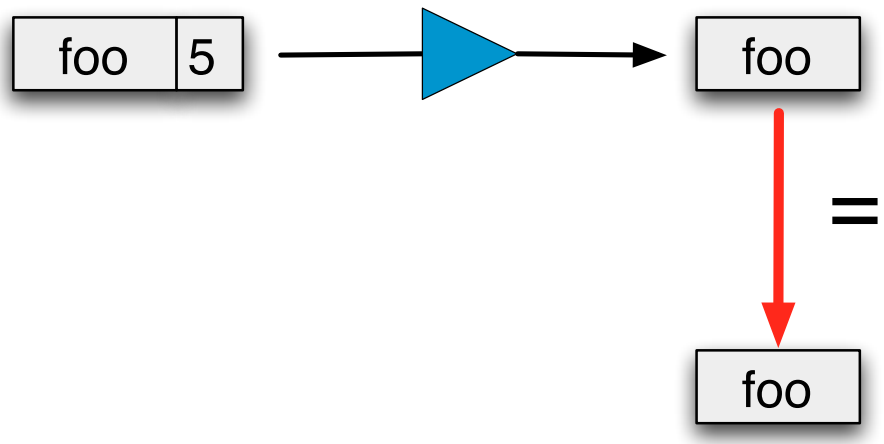
Another Unreasonable Example



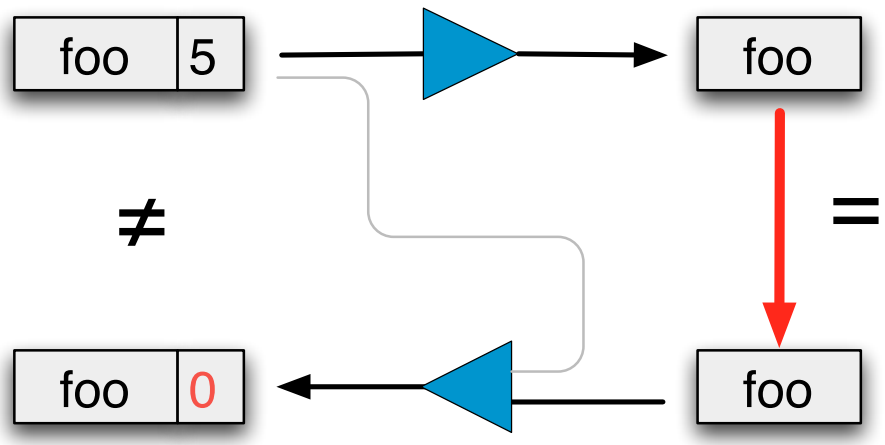
Another Unreasonable Example



Another Unreasonable Example



Another Unreasonable Example



The GetPut law

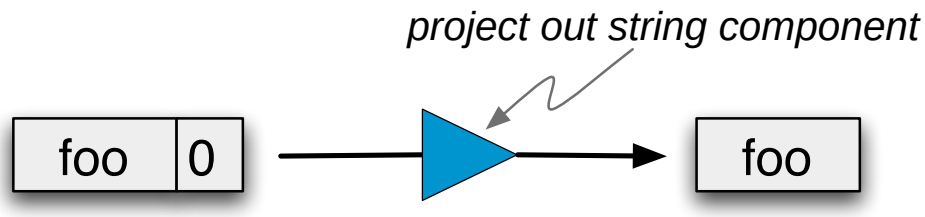
Principle:

If the view does not change, neither should the source.

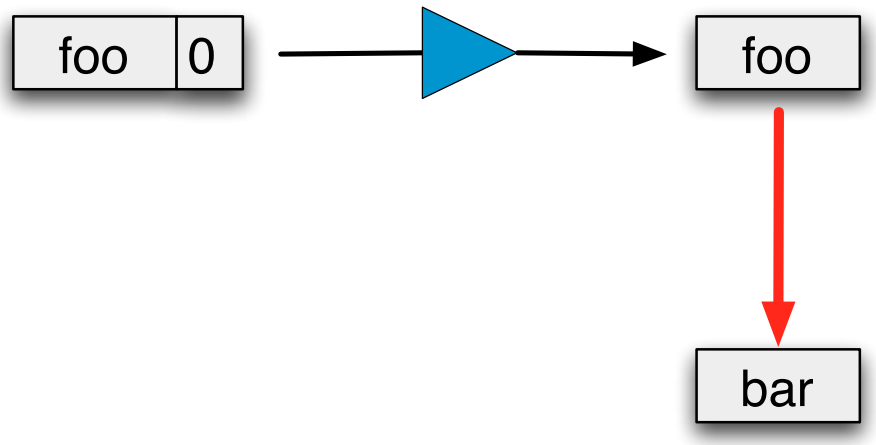
Formally:

$$\textbf{put}\ (\textbf{get}\ s)\ s \quad = \quad s$$

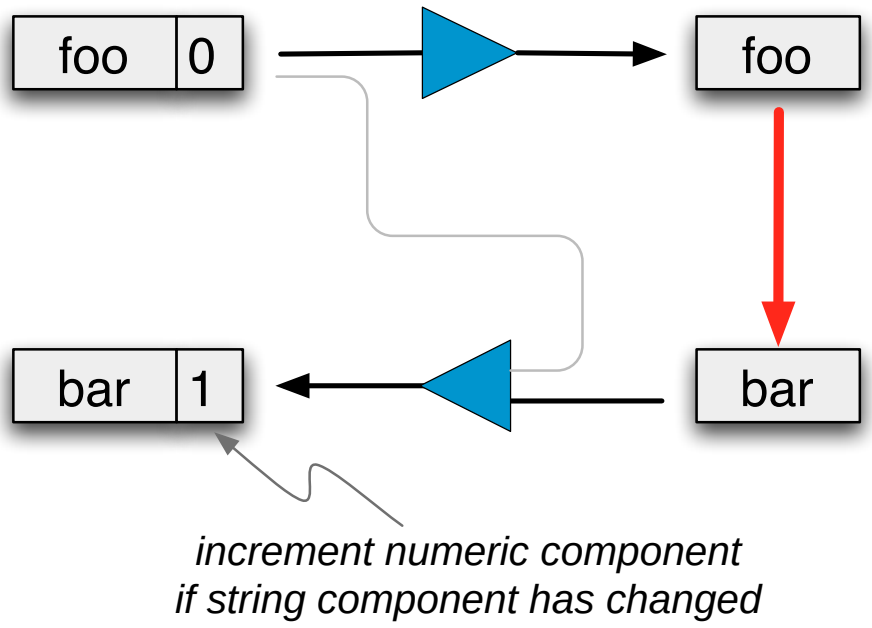
Another Debatable Example



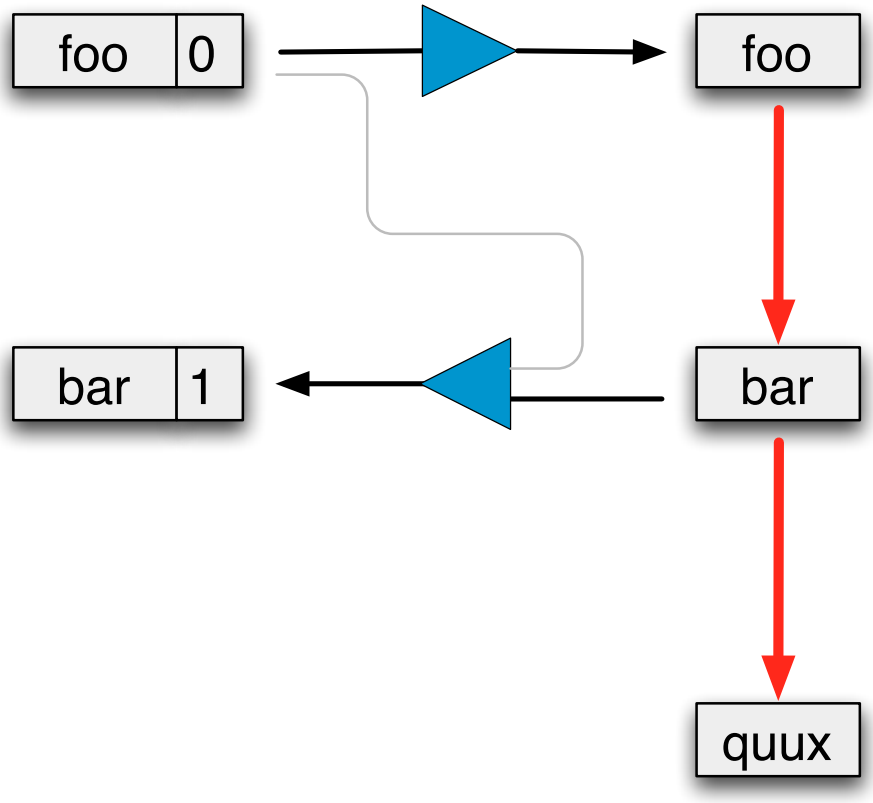
Another Debatable Example



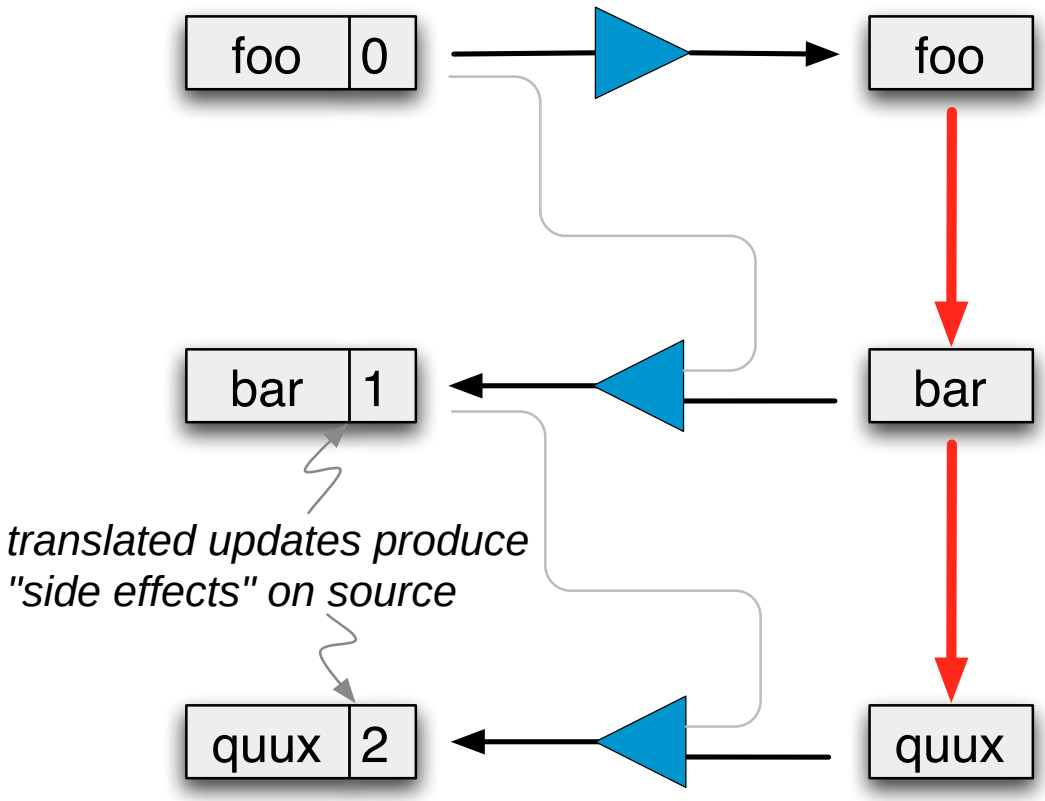
Another Debatable Example



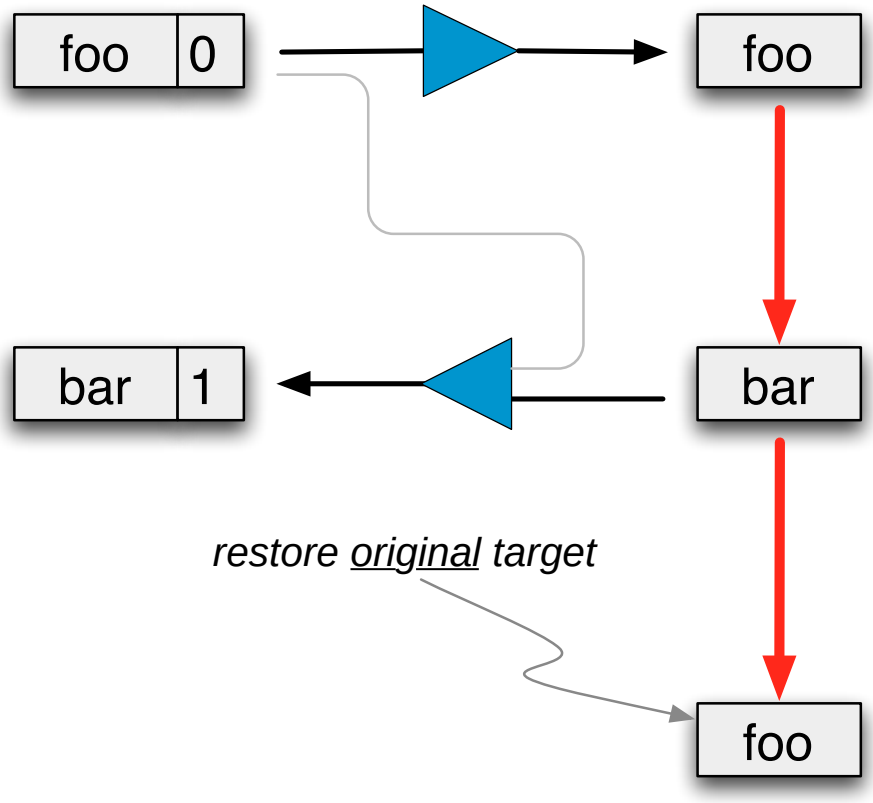
Another Debatable Example



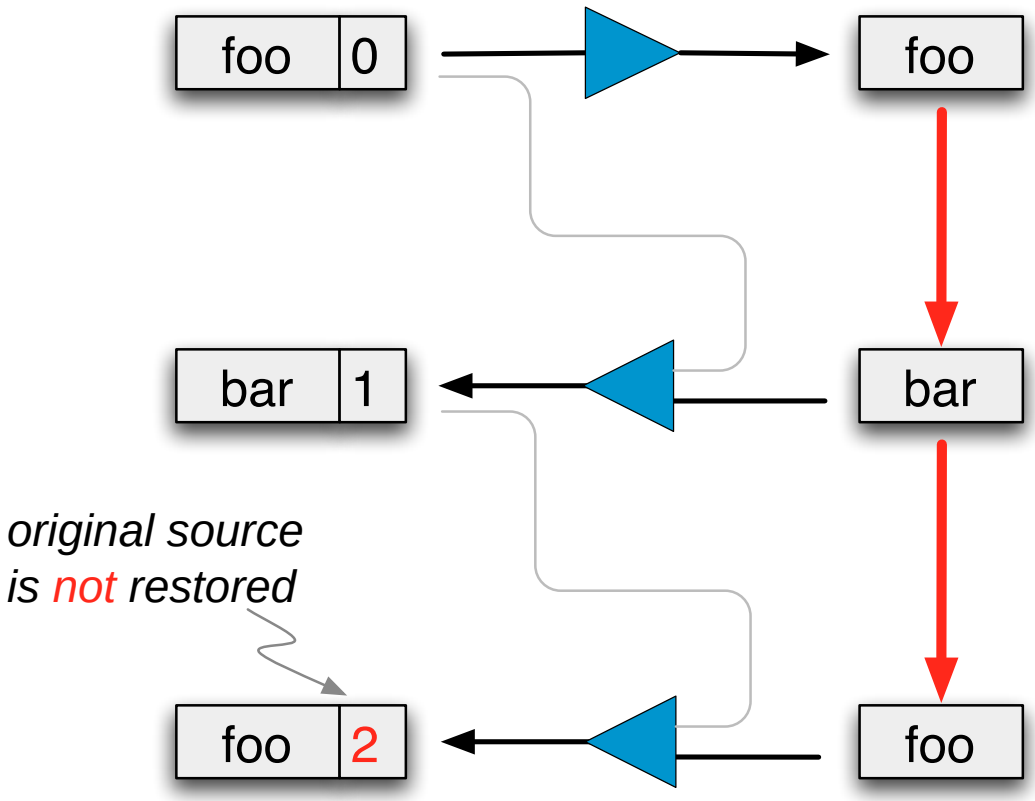
Another Debatable Example



Another Debatable Example



Another Debatable Example



The PutPut law

Principle:

Each update should completely overwrite the effect of the previous one. In particular, the effect of two puts in a row should be the same as just the second.

Formally:

$$\textbf{put}\ v_2\ (\textbf{put}\ v_1\ s) \quad = \quad \textbf{put}\ v_2\ s$$

The PutPut law

Principle:

Each update should completely overwrite the effect of the previous one. In particular, the effect of two puts in a row should be the same as just the second.

Formally:

$$\textbf{put}\ v_2\ (\textbf{put}\ v_1\ s) \quad = \quad \textbf{put}\ v_2\ s$$

Nice properties:

- Ensures that every update can be "rolled back"
- Implies that S is isomorphic to $V \times C$, for some C
- Bancilhon and Spyratos's update translators preserving a "constant complement" are a slight refinement

The PutPut law

Principle:

Each update should completely overwrite the effect of the previous one. In particular, the effect of two puts in a row should be the same as just the second.

Formally:

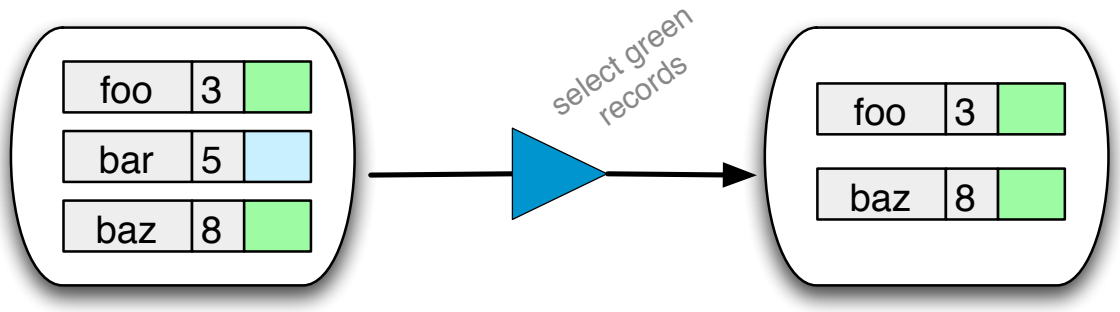
$$\textbf{put}\ v_2\ (\textbf{put}\ v_1\ s) \quad = \quad \textbf{put}\ v_2\ s$$

Nice properties:

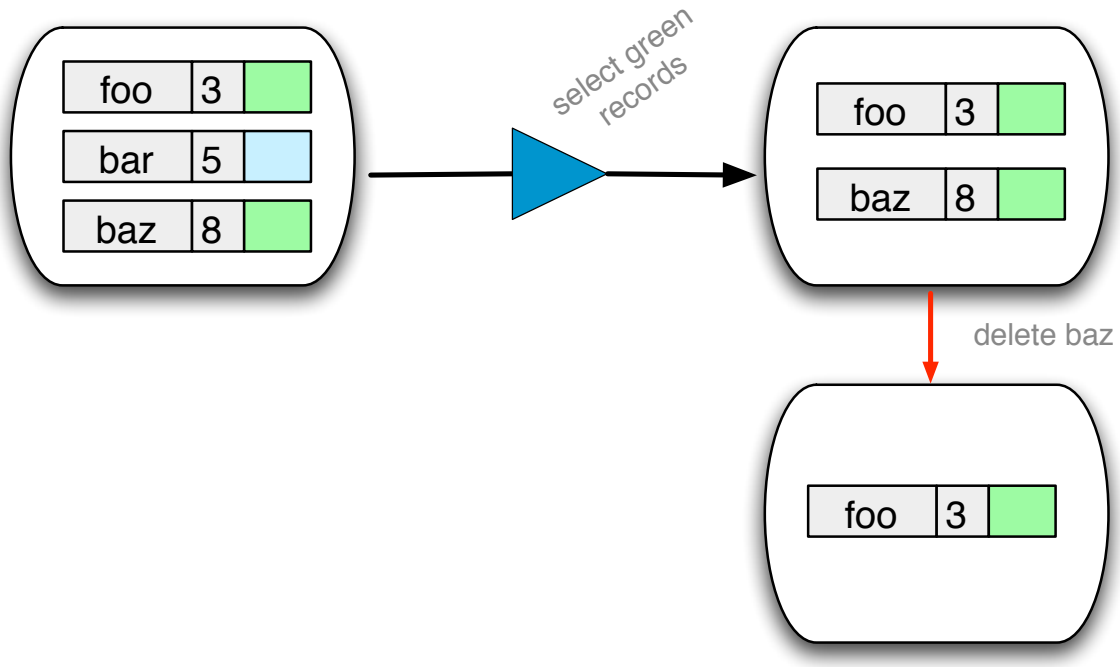
- Ensures that every update can be "rolled back"
- Implies that S is isomorphic to $V \times C$, for some C
- Bancilhon and Spyratos's update translators preserving a "constant complement" are a slight refinement

Seems sensible. But do we want to always require it?

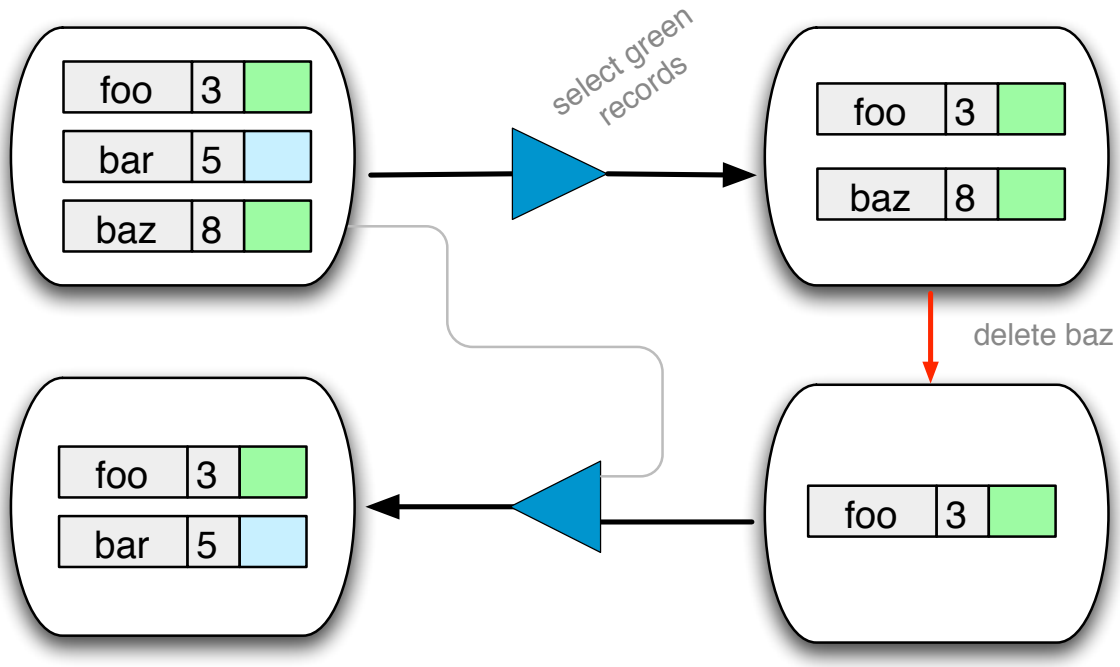
Another Example



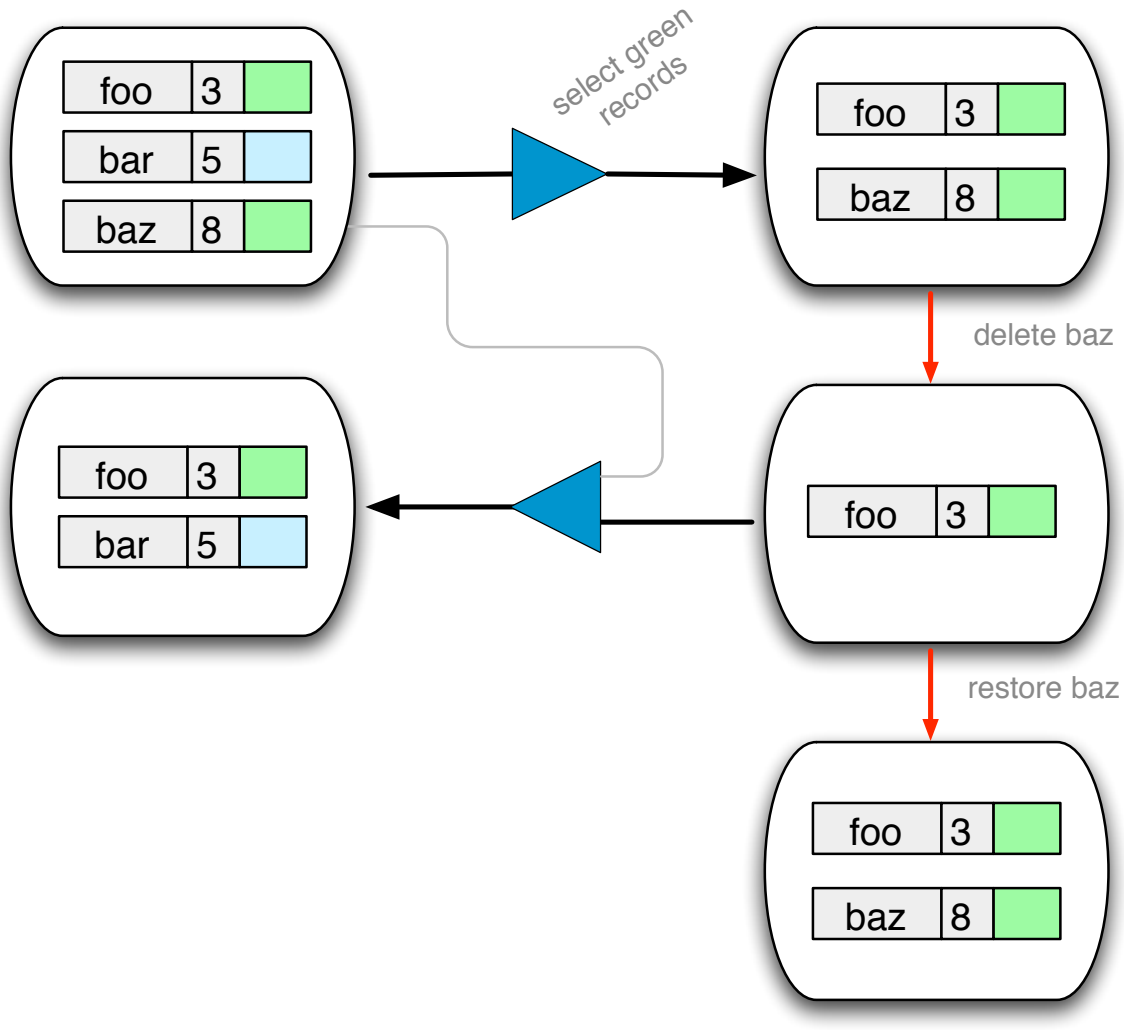
Another Example



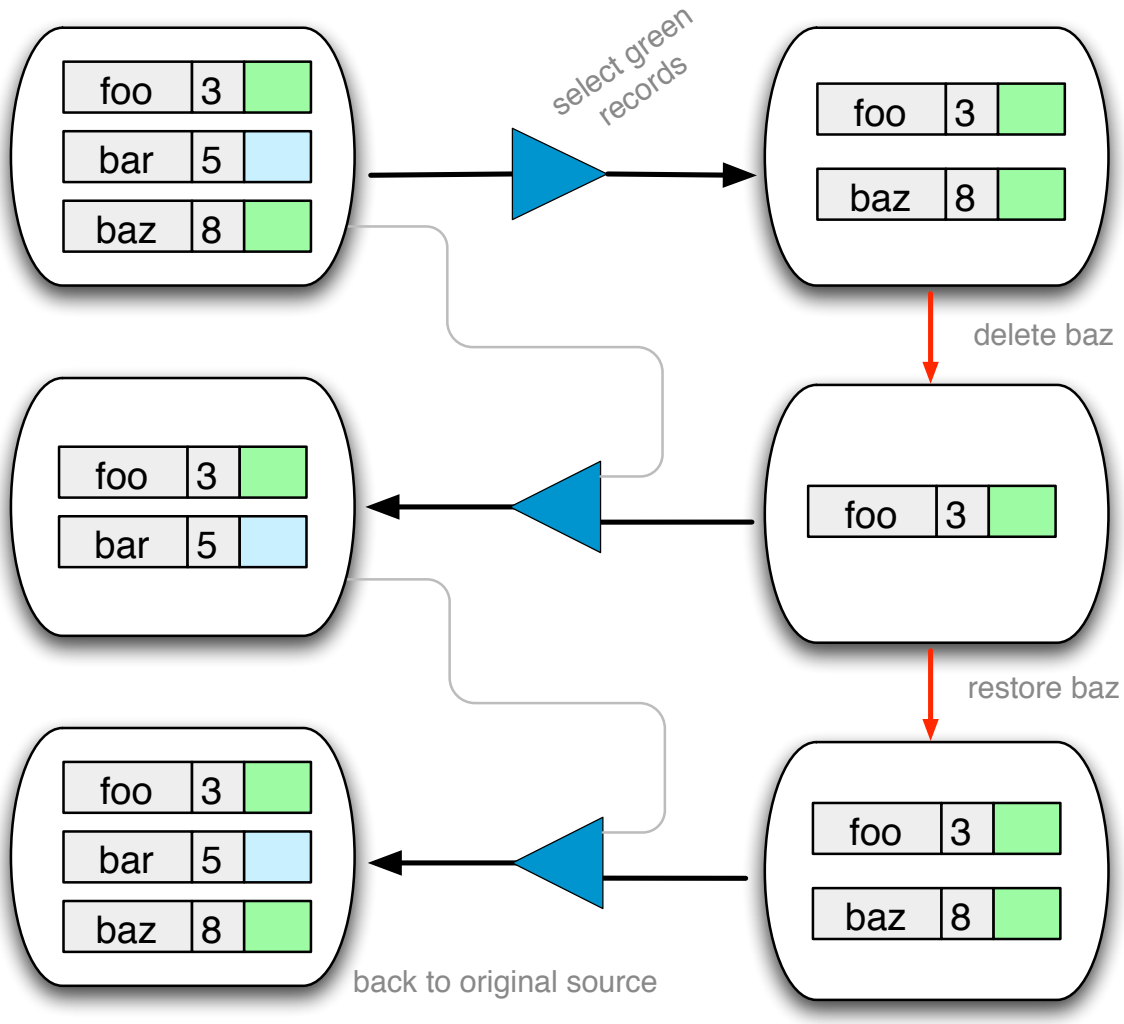
Another Example



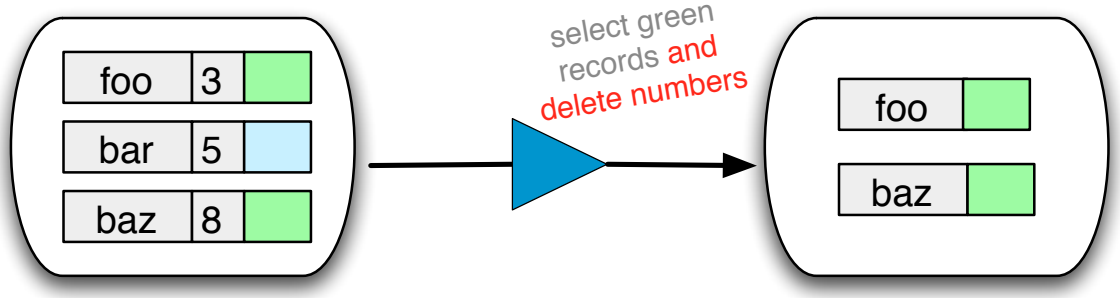
Another Example



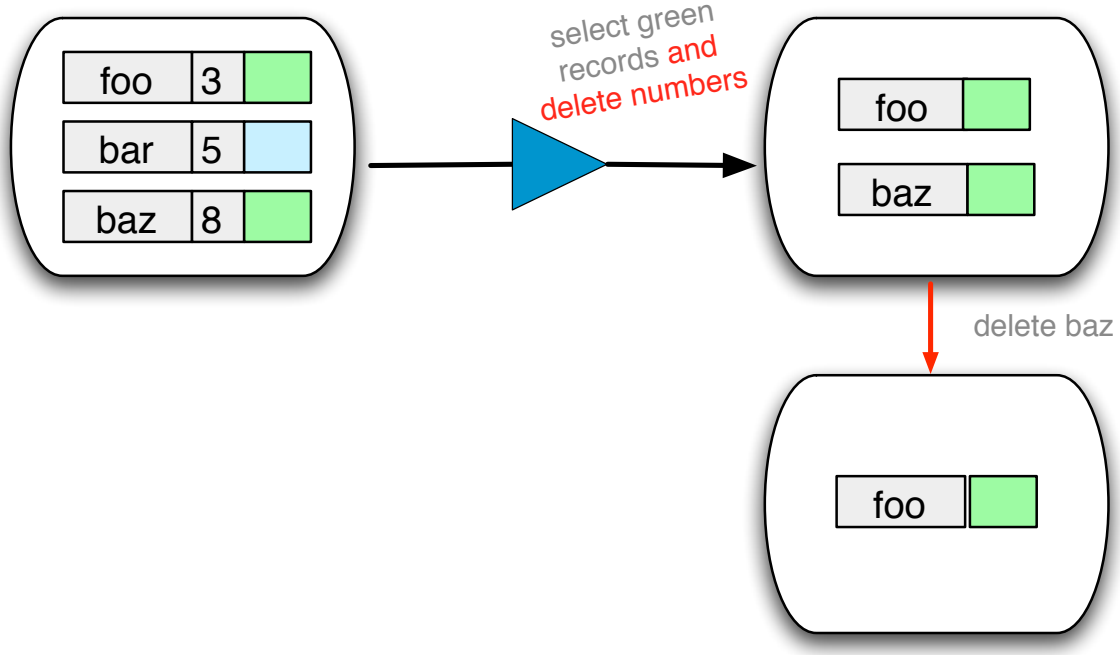
Another Example



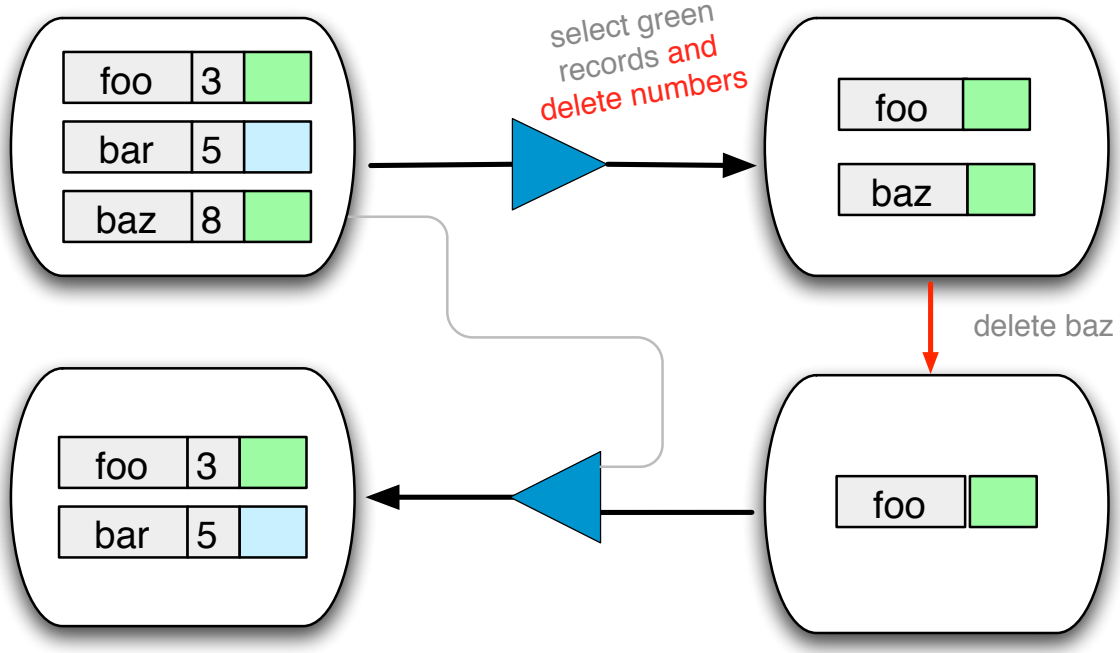
Yet Another Example



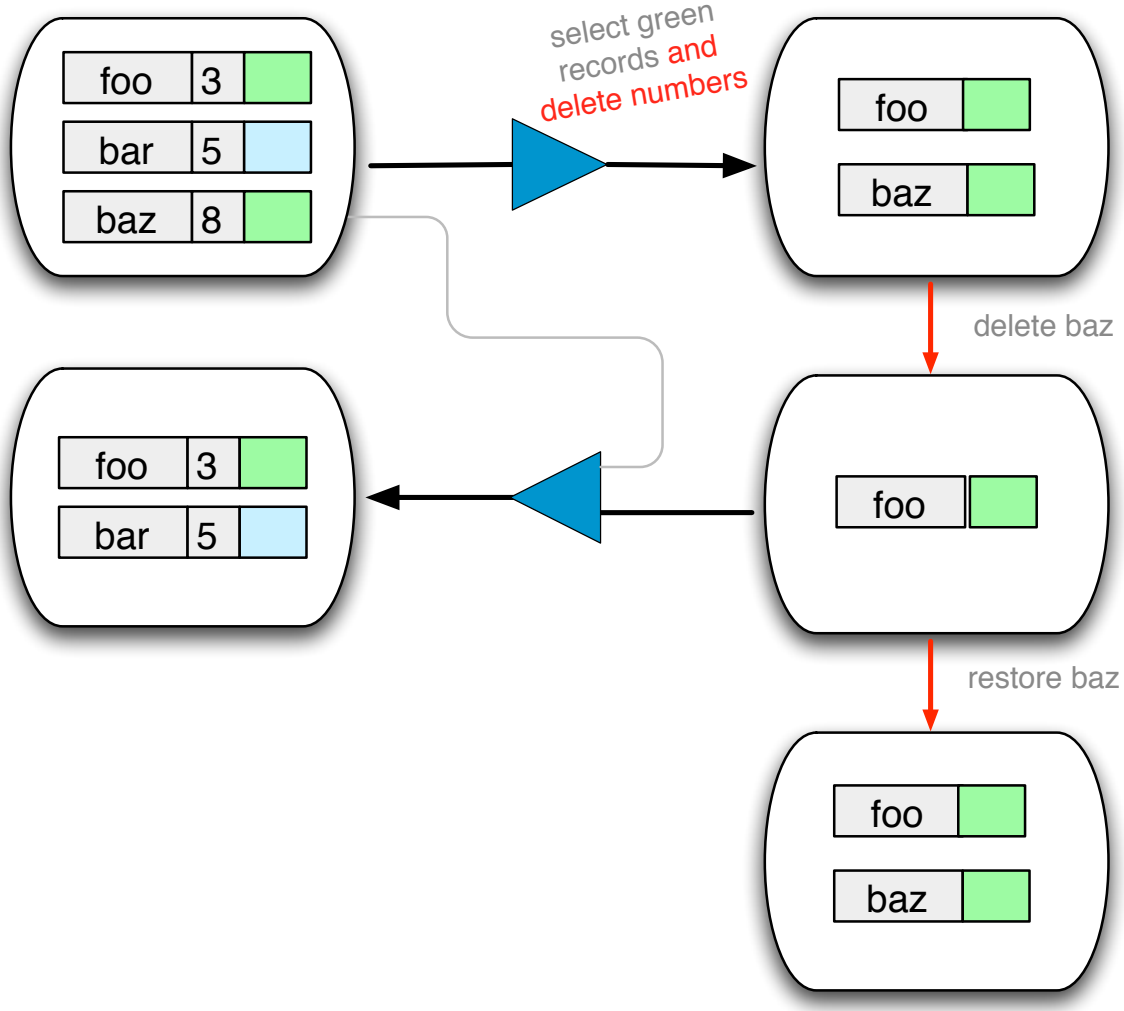
Yet Another Example



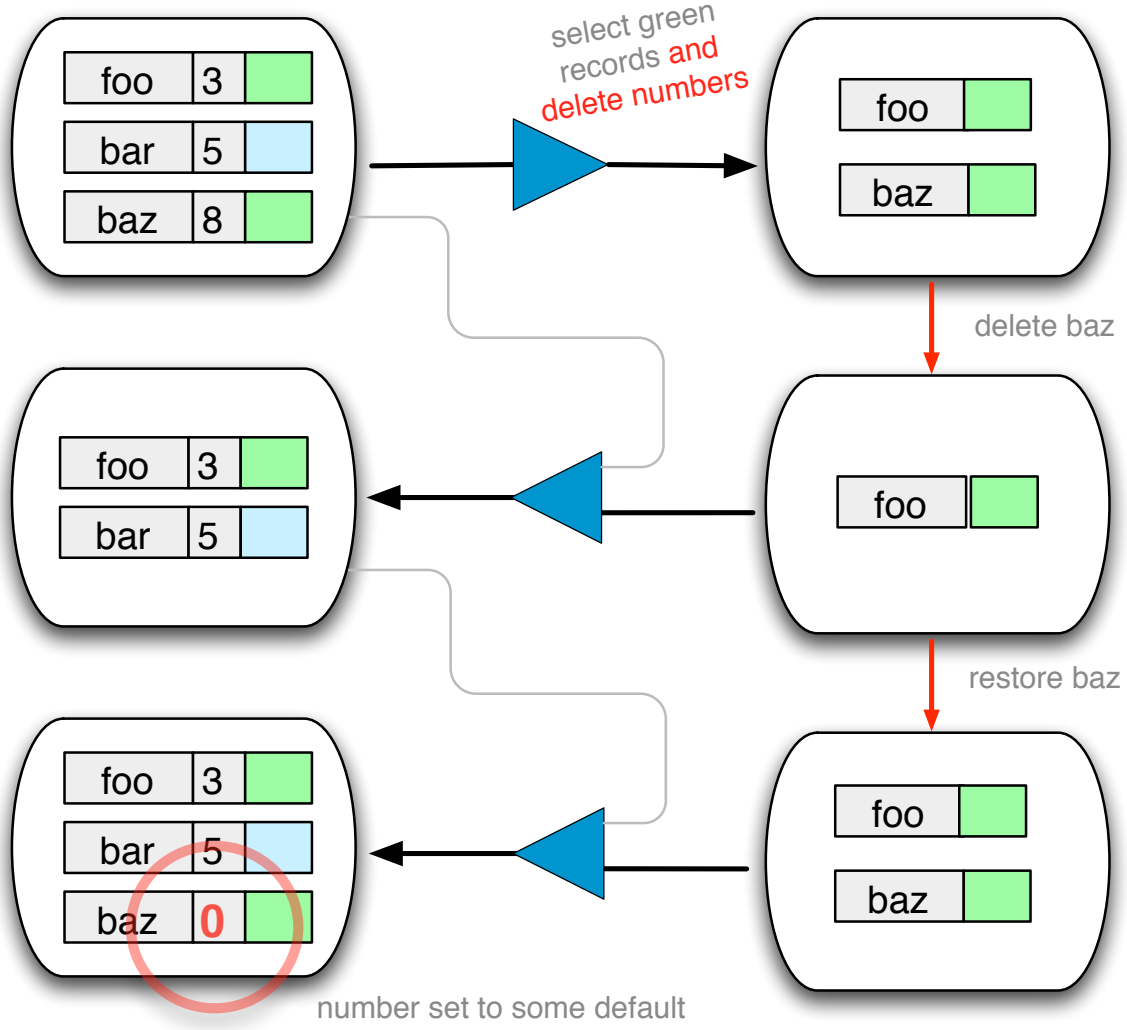
Yet Another Example



Yet Another Example



Yet Another Example



GetPut vs. PutPut

The GetPut law implies a weaker variant of PutPut:

$$\textbf{put}\ v\ (\textbf{put}\ v\ s) \quad = \quad \textbf{put}\ v\ s$$

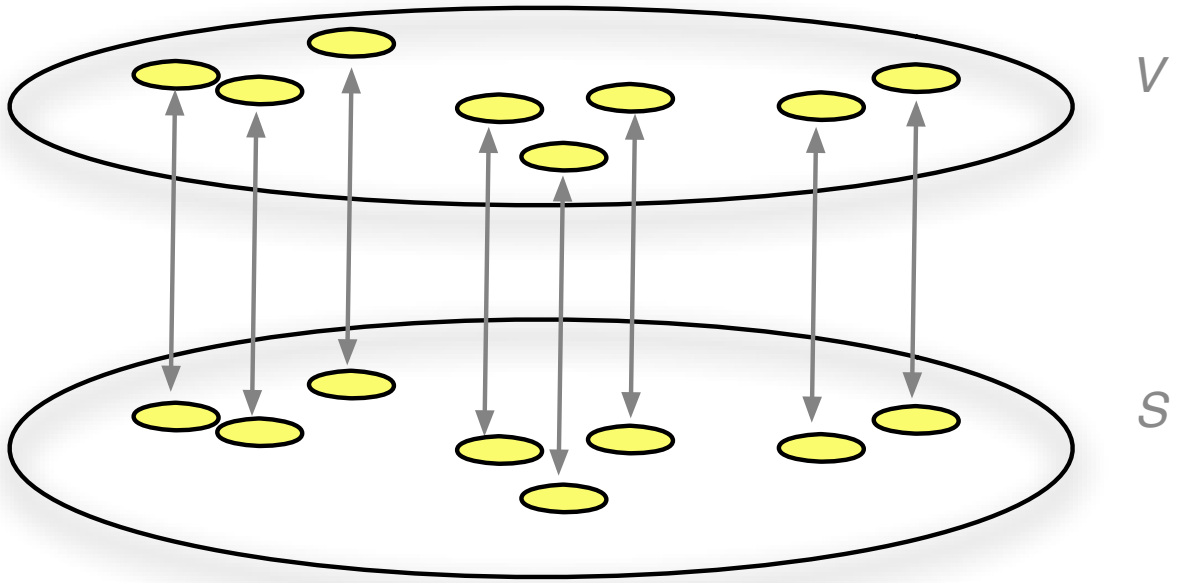
Proof is a straightforward calculation:

$$\begin{array}{lll} & \textbf{put}\ v\ (\textbf{put}\ v\ s) & \\ = & \textbf{put}\ (\textbf{get}\ (\textbf{put}\ v\ s))\ (\textbf{put}\ v\ s) & \text{by PutGet} \\ = & \textbf{put}\ v\ s & \text{by GetPut} \end{array}$$

Question #4:

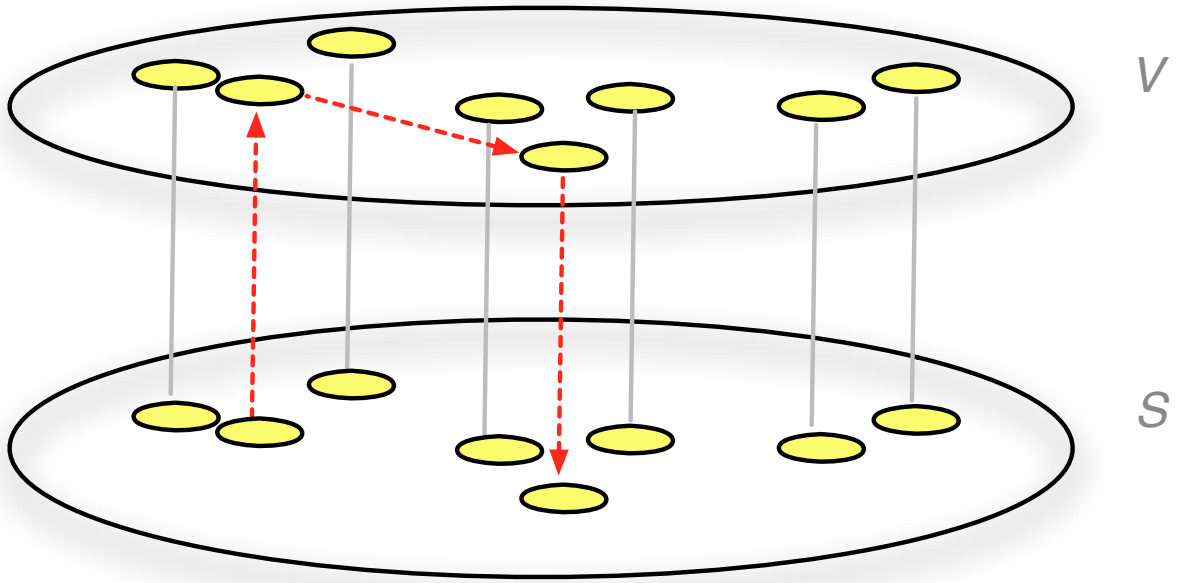
Given a **get** *function, can programmers choose an appropriate* **put** *function?*

How many **puts**? (Bijective Case)



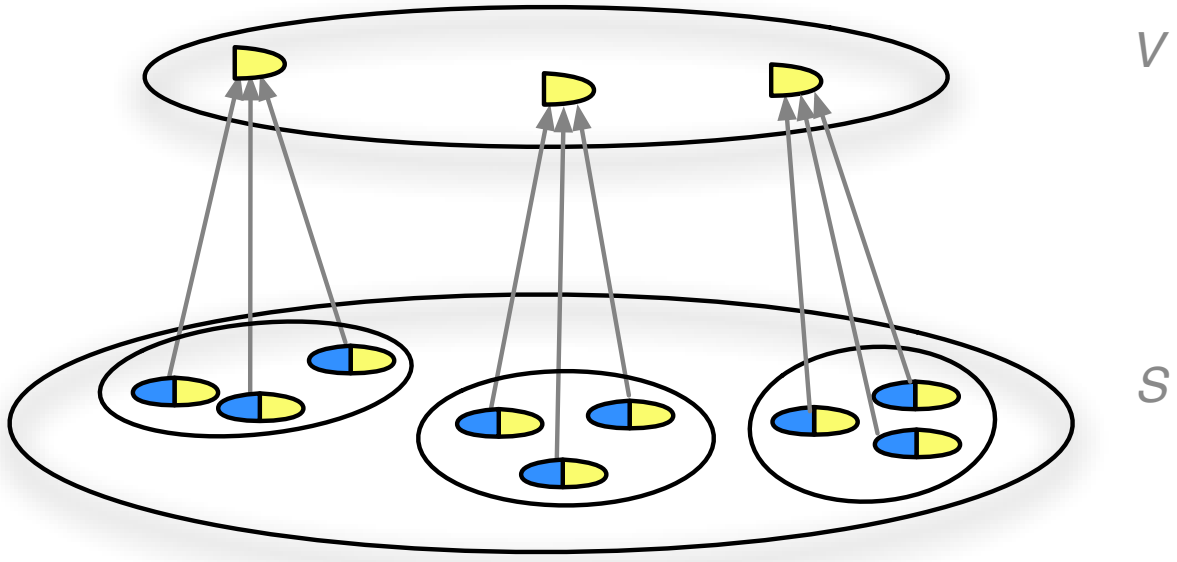
A bijective lens defines a one-to-one correspondence between *S* and *V*.

How many **puts**? (Bijective Case)



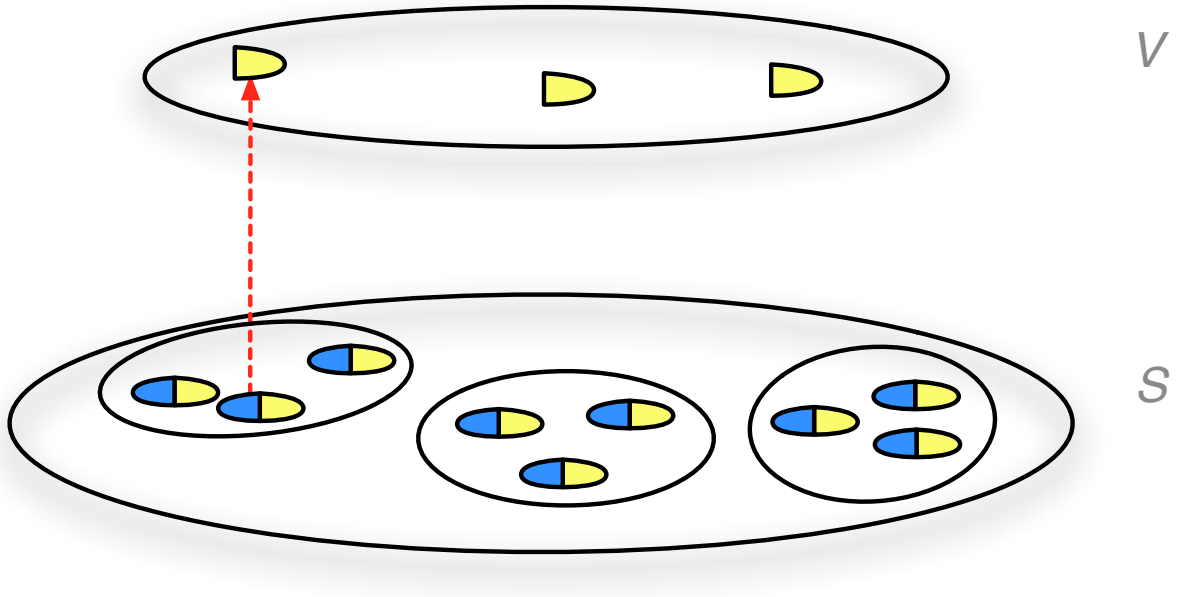
The behavior of **put** is completely fixed by **get**.

How many **puts**? (Bidirectional Case)



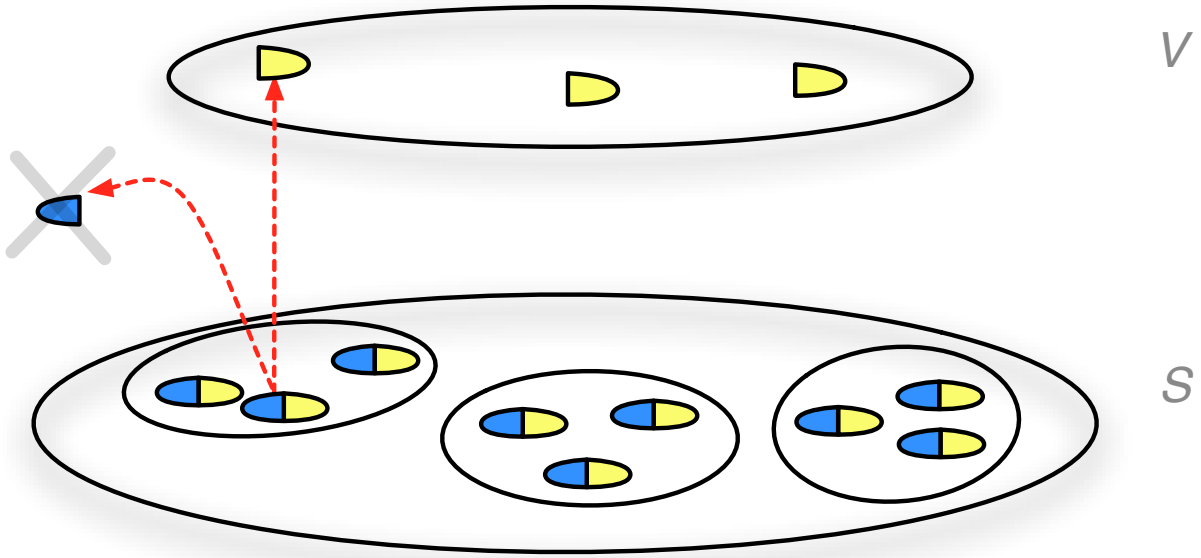
If we are defining a bidirectional transformation, then many structures from *S* can map onto the same structure from *V*.

How many **puts**? (Bidirectional Case)



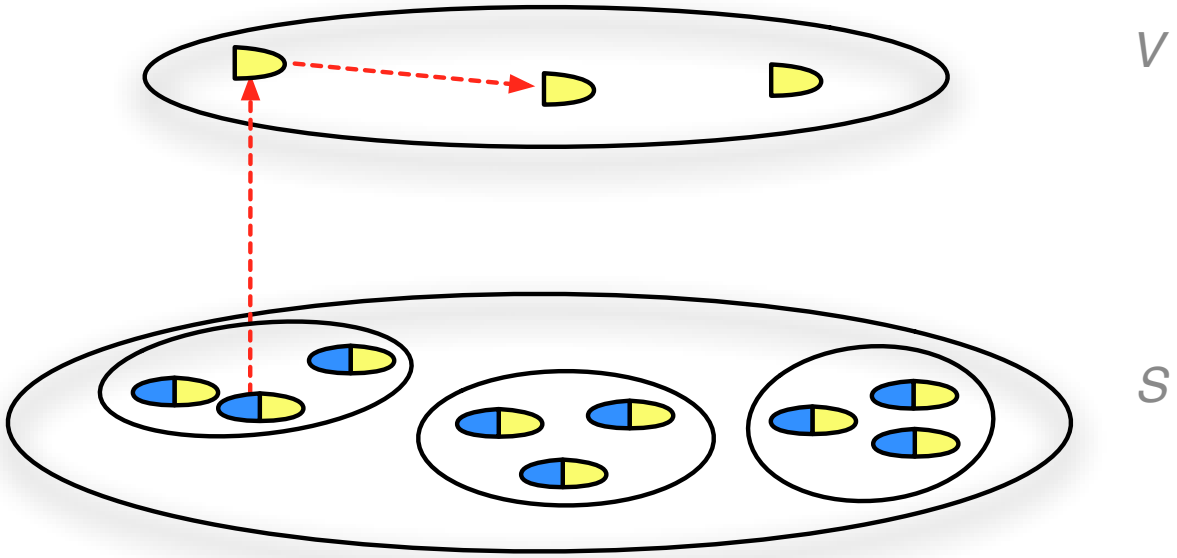
The **get** function projects out part of the information in the source structure…

How many **puts**? (Bidirectional Case)



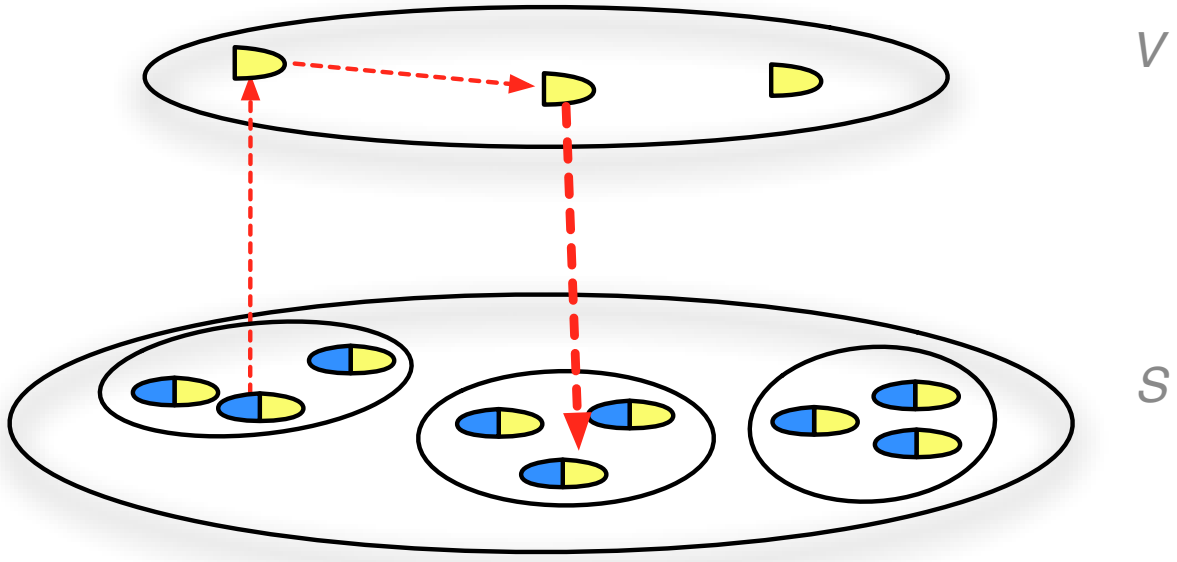
The **get** function projects out part of the information in the source structure… and throws away the rest.

How many **puts**? (Bidirectional Case)



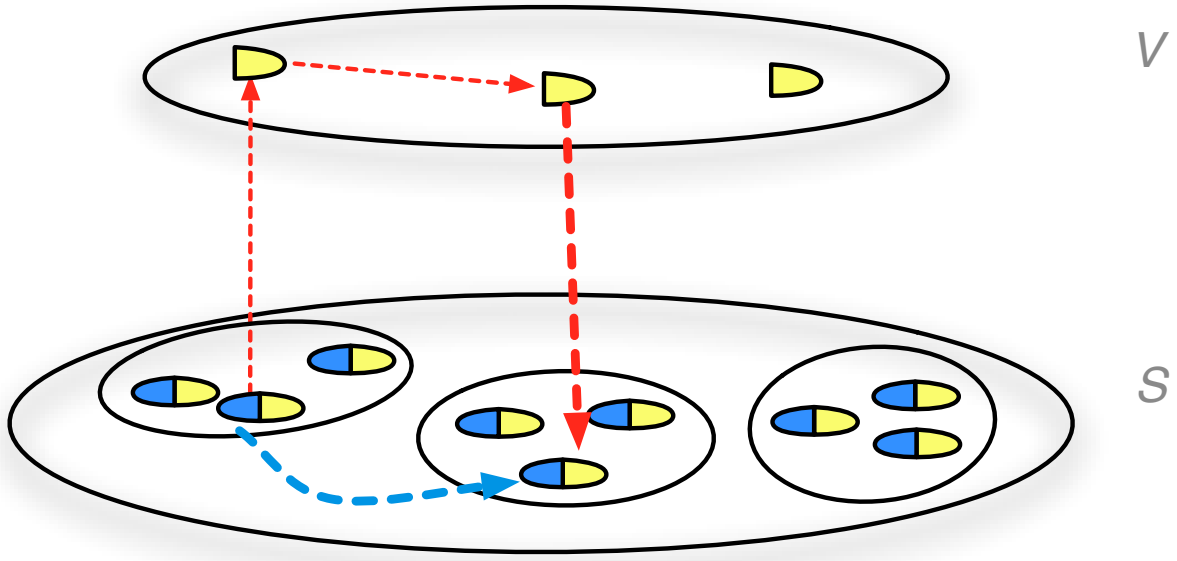
If the lens obeys PutPut…

How many **puts**? (Bidirectional Case)



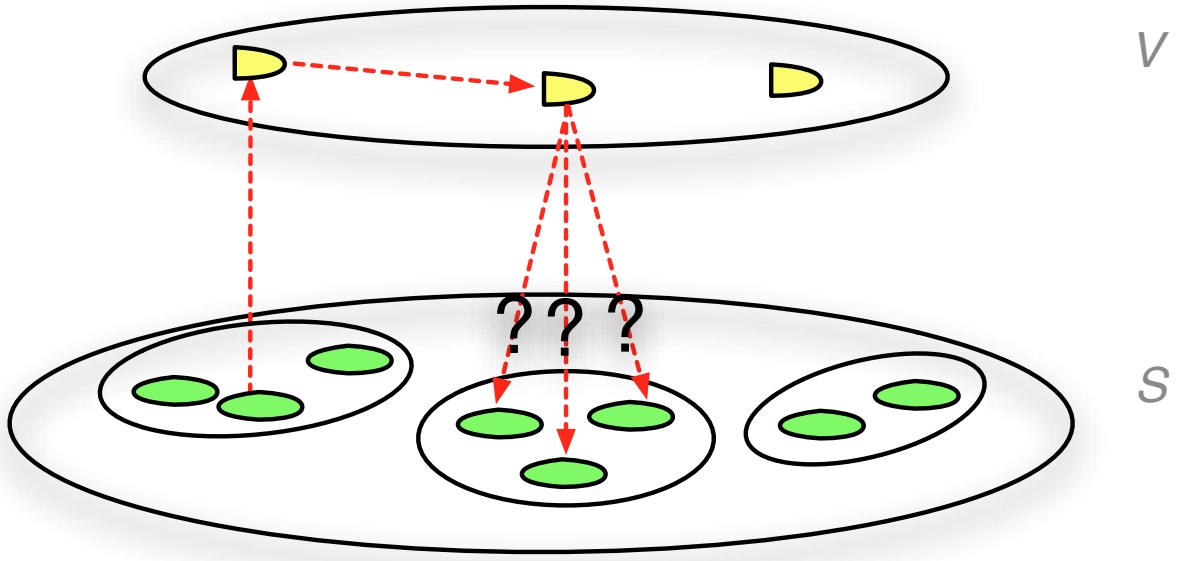
If the lens obeys PutPut… then the "view part" of the new source structure is fixed by PutGet…

How many **puts**? (Bidirectional Case)



If the lens obeys PutPut… then the "view part" of the new source structure is fixed by PutGet… and the "projected away part" is fixed by PutPut to be exactly the one from the original source.

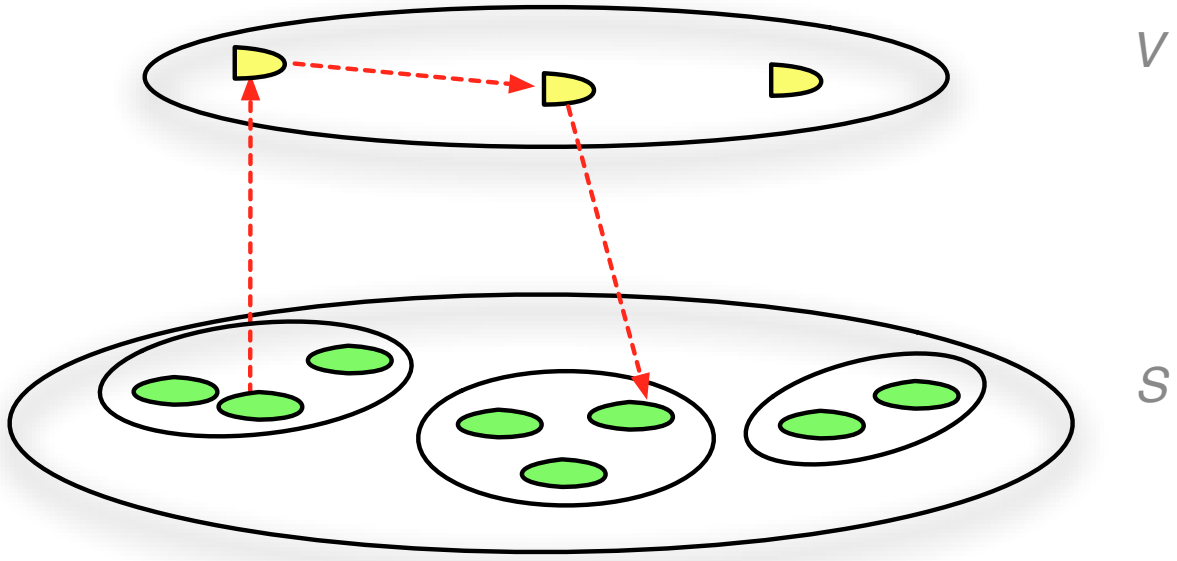
How many **puts**? (Bidirectional Case)



However, if the lens only obeys PutGet, then the behavior of **put** is less constrained...

...and there are many **puts** to choose from!

How many **puts**? (Bidirectional Case)



However, if the lens only obeys PutGet, then the behavior of **put** is less constrained...

...and there are many **put**s to choose from!

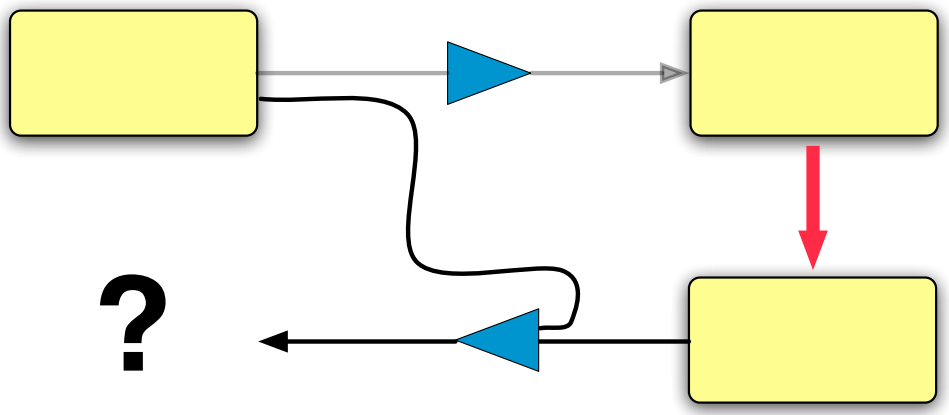
Need extra information to select one.

Question #5:

Does **put** *handle every update?*

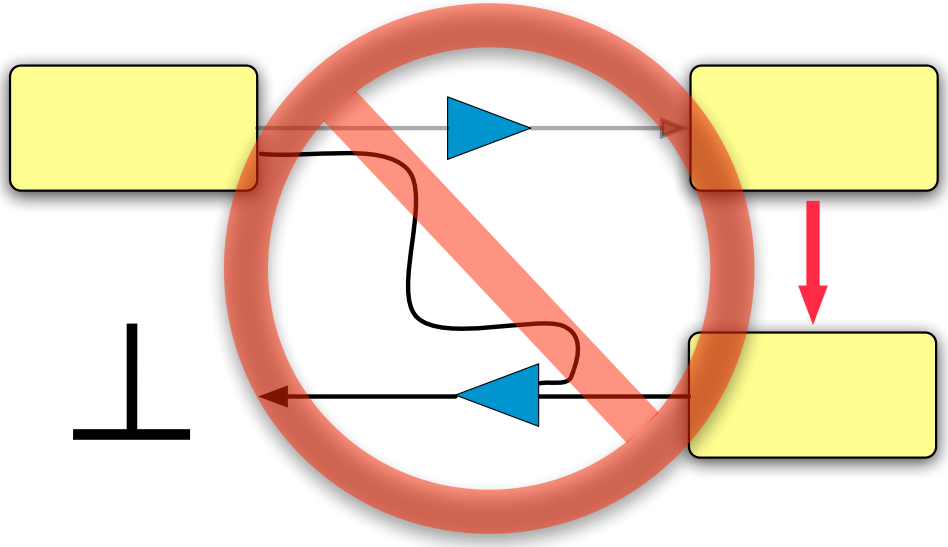
Totality

Does the **put** function handle every view and every source or does it reject some combinations (by failing)?



Totality

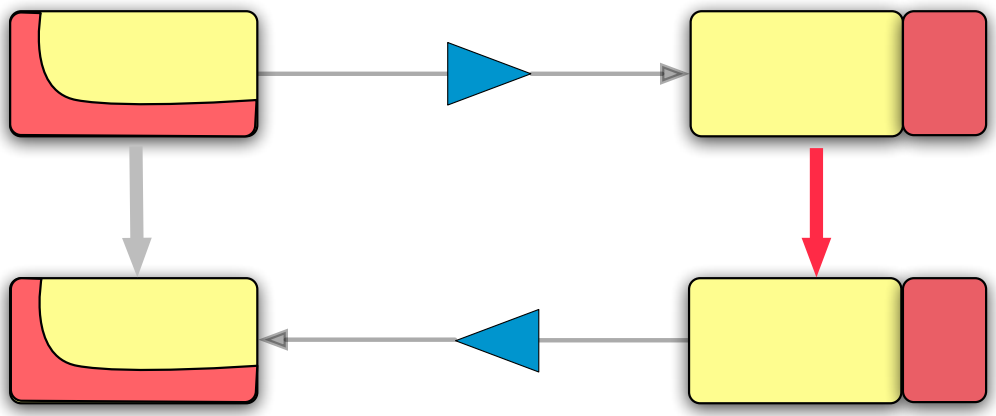
Does the **put** function handle every view and every source or does it reject some combinations (by failing)?



Totality ensures that the view is a robust abstraction of the source [Hegner '90]

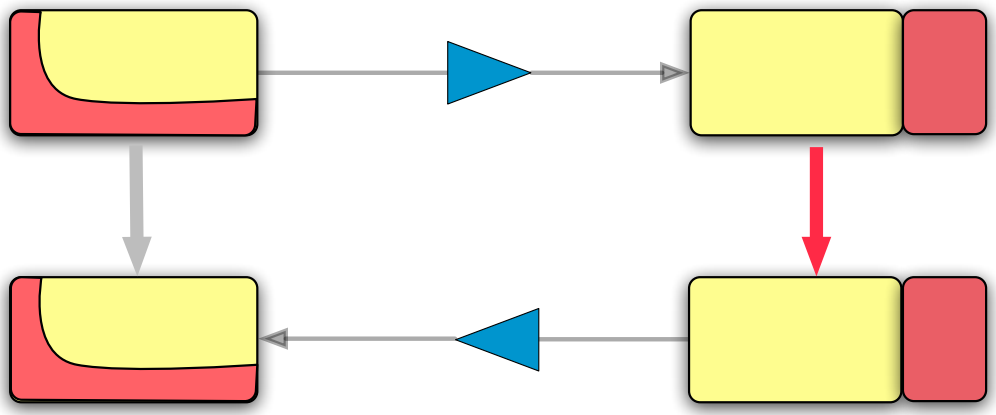
Totality and Injective Embeddings

Can simulate a bidirectional transformation with an injective **get** function by storing a complement along with the view [Hu, Mu, Takeichi '04]



Totality and Injective Embeddings

Can simulate a bidirectional transformation with an injective **get** function by storing a complement along with the view [Hu, Mu, Takeichi '04]



However, in general, the **put** function will only be defined on $\{(v, c) \in V \times C \mid \exists s.\ \mathbf{get}\ s = (v, c)\}$ and *not* $V \times C$.

Summary

1. What is an update?
2. Bijective or bidirectional? Symmetric?
3. Reasonable?
4. Choice of **put**?
5. Total?