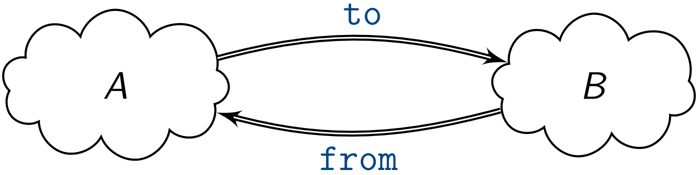
Bidirectional Transformations

—

a PL perspective

BIRS meeting on BX, 2013

Bidirectional Transformations (BX)



database source ⇔ materialized view

software model ⇔ code

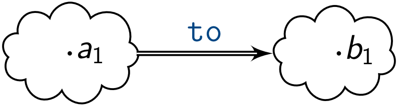
document representation ⇔ screen visualization

concrete syntax ⇔ abstract syntax

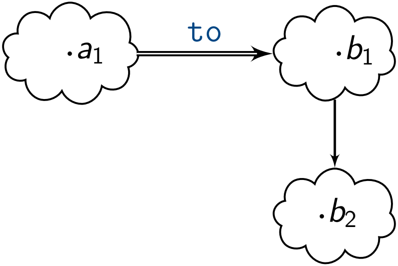
abstract datatype ⇔ actual implementation

program input ⇔ program output

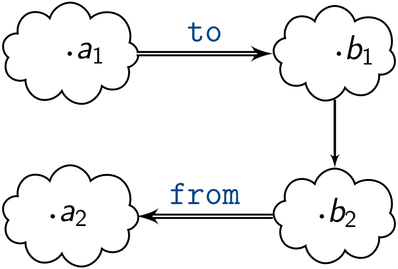
Bidirectional Transformations



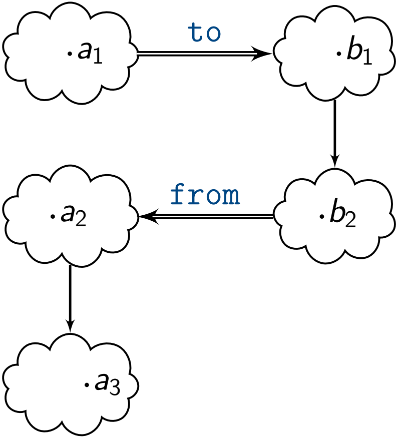
Bidirectional Transformations



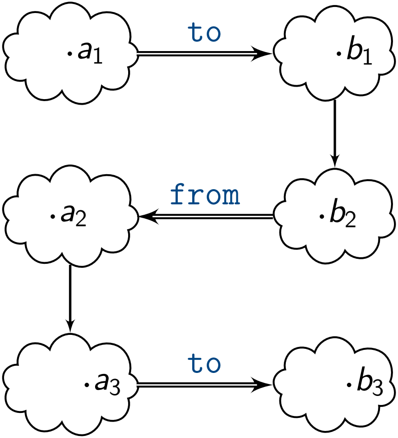
Bidirectional Transformations



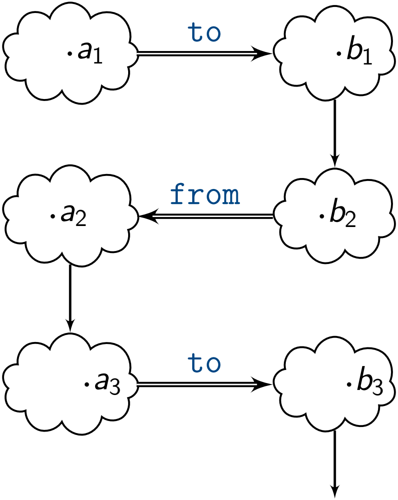
Bidirectional Transformations



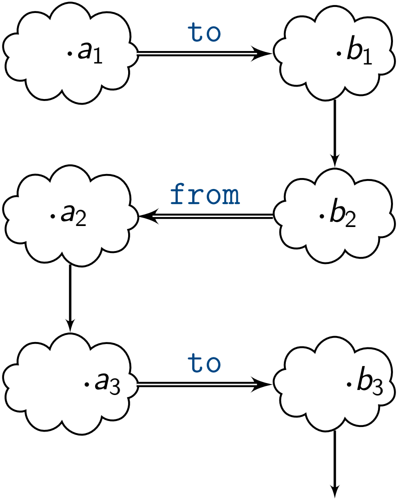
Bidirectional Transformations



Bidirectional Transformations

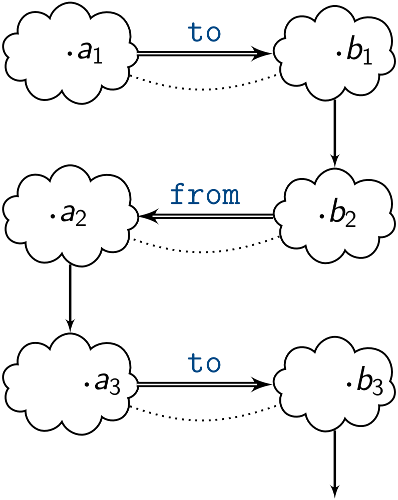


Bidirectional Transformations



unless bijective, typically additional information needed/useful

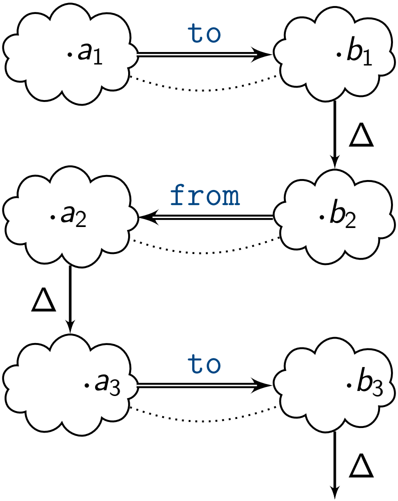
Bidirectional Transformations



unless bijective, typically additional information needed/useful:

- about connections between A and B (objects)

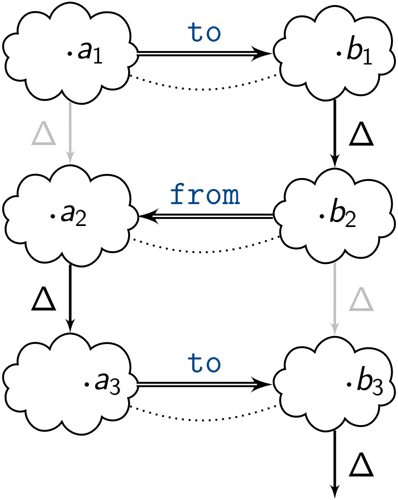
Bidirectional Transformations



unless bijective, typically additional information needed/useful:

- about connections between A and B (objects)
- about the updates on either side

Bidirectional Transformations



unless bijective, typically additional information needed/useful:

- about connections between A and B (objects)
- about the updates on either side

Objectives for this Talk

- get everybody into “BX mode” for the week
- set out basic premises of the PL approach, paradigmatic problems
- introduce terminology and semantic principles
- no details of specific solutions
- relate to what “we” think is solved and what not
- open discussion

What’s specific about “the PL approach”, anyway?

What's specific about "the PL approach", anyway?

- focus on the transformations/functions themselves, not so much on the data

What's specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws

What's specific about "the PL approach", anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)

What's specific about "the PL approach", anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)

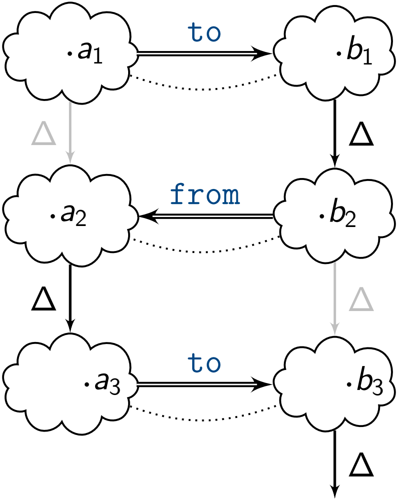
What's specific about "the PL approach", anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)
- being driven by our favourite new PL techniques

What’s specific about “the PL approach”, anyway?

- focus on the transformations/functions themselves, not so much on the data
- focus on extensional semantics and laws
- correctness by construction/derivation (as opposed to a-posteriori verification)
- assuming a very clean setting (naive?)
- being driven by our favourite new PL techniques
- typically, algebraic data domains

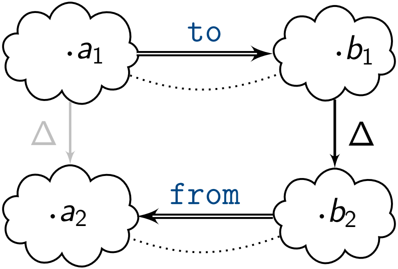
Bidirectional Transformations



unless bijective, typically additional information needed/useful:

- about connections between A and B (objects)
- about the updates on either side

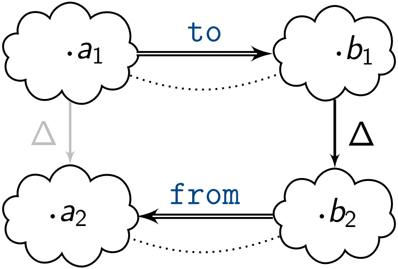
Bidirectional Transformations



focus on:

- single-side updates
- one-step updates

Bidirectional Transformations



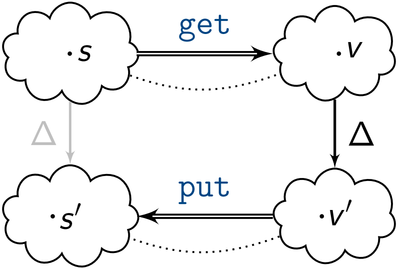
focus on:

- single-side updates
- one-step updates

also, focus on asymmetric setting:

- `to` usually non-injective, henceforth called `get`
- `from` then called `put`, definitely needs extra info
- for simplicity, state-based
- “sources” and “views”

Bidirectional Transformations



focus on:

- single-side updates
- one-step updates

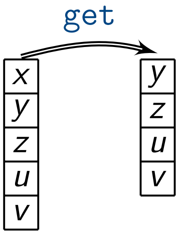
also, focus on asymmetric setting:

- `to` usually non-injective, henceforth called `get`
- `from` then called `put`, definitely needs extra info
- for simplicity, state-based
- “sources” and “views”

Bidirectional Transformations

A closer look at representing $\cdot\overset{s}{\cdots\cdots}\cdot v$ connections.

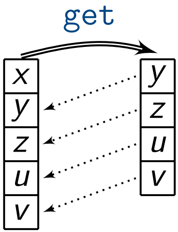
For example:



Bidirectional Transformations

A closer look at representing $s \cdots v$ connections.

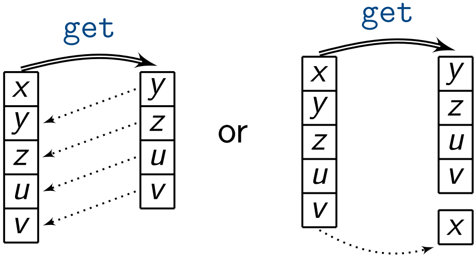
For example:



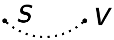
Bidirectional Transformations

A closer look at representing $\overset{s}{\cdot} \cdots \overset{v}{\cdot}$ connections.

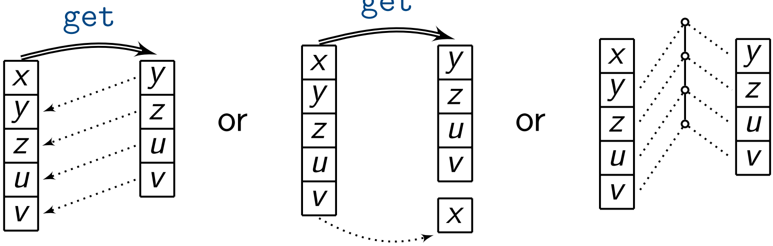
For example:



Bidirectional Transformations

A closer look at representing  connections.

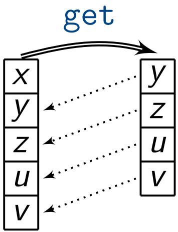
For example:



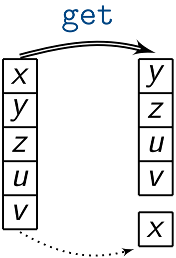
Bidirectional Transformations

A closer look at representing s ⋯ v connections.

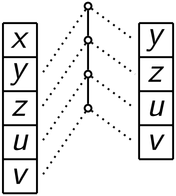
For example:



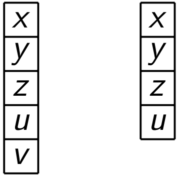
or



or



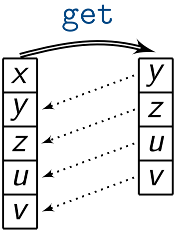
Why is it not enough to look just at the data?



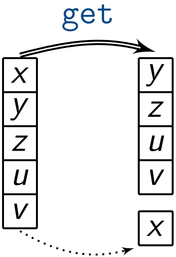
Bidirectional Transformations

A closer look at representing $s \cdots v$ connections.

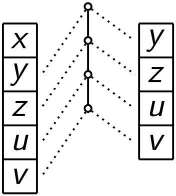
For example:



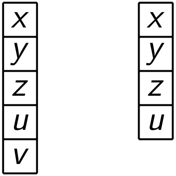
or



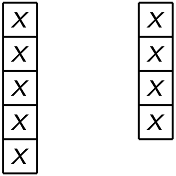
or



Why is it not enough to look just at the data?



Because of:



Bidirectional Transformations

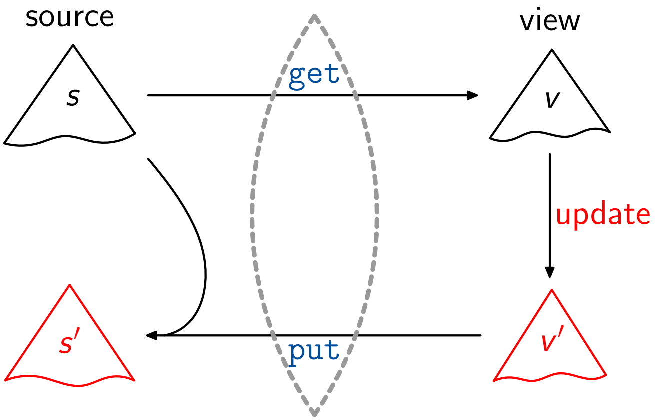
Some further relevant aspects:

- What artifacts need to be specified?
    - both `get` and `put`
    - only one of them, the other derived
    - a more abstract artifact, from which both derivable
- How are they specified, manipulated, analyzed?
- What properties are they expected to have?
- What influence does a user, modeller, programmer have?

Properties / Laws

Bidirectional Transformations

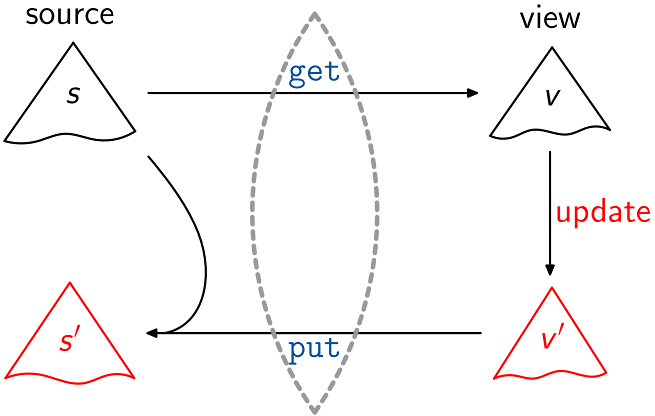
Specific asymmetric setting, state-based:



$$\texttt{get} :: S \rightarrow V$$
$$\texttt{put} :: S \rightarrow V \rightarrow S$$

Bidirectional Transformations

Specific asymmetric setting, state-based:

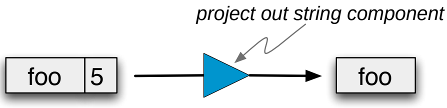


$\texttt{get} :: S \rightarrow V$

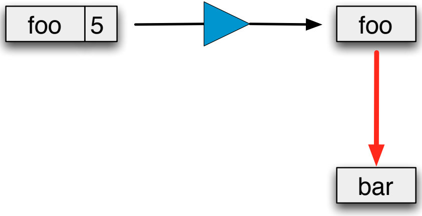
$\texttt{put} :: S \rightarrow V \rightarrow S$

assumed total!

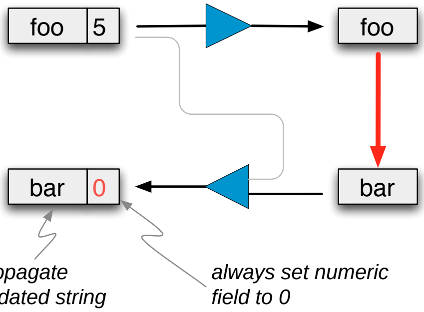
About Behavior under No-Change



About Behavior under No-Change



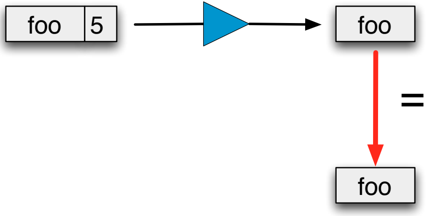
About Behavior under No-Change



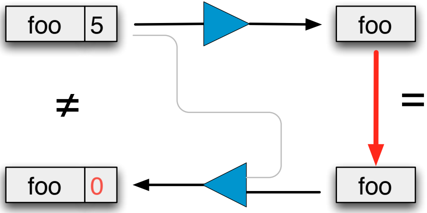
propagate updated string

always set numeric field to 0

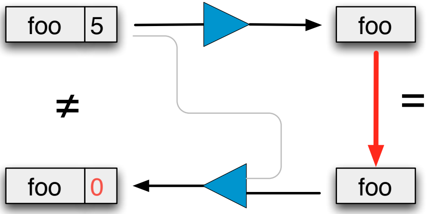
About Behavior under No-Change



About Behavior under No-Change



About Behavior under No-Change

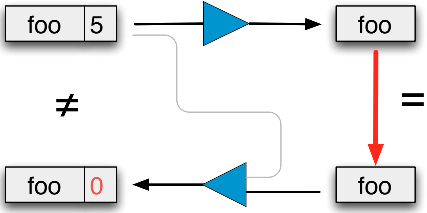


Principle: If the view does not change, neither should the source.

To prevent this, the GetPut law:

$$\texttt{put}\ s\ (\texttt{get}\ s) = s$$

About Behavior under No-Change



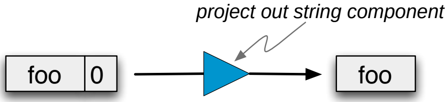
Principle: If the view does not change, neither should the source.

To prevent this, the GetPut law:

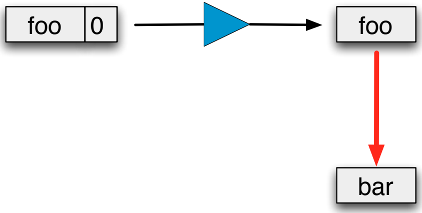
$$\mathtt{put}\ s\ (\mathtt{get}\ s) = s$$

NB: For this, `put` must be surjective.

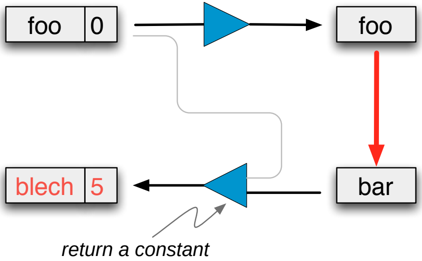
About Preservation of Changes



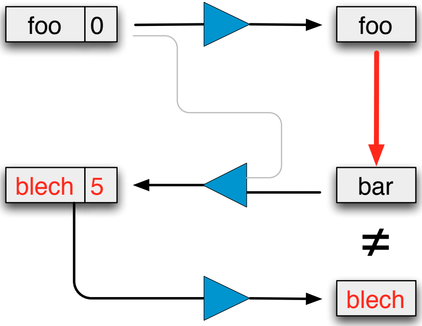
About Preservation of Changes



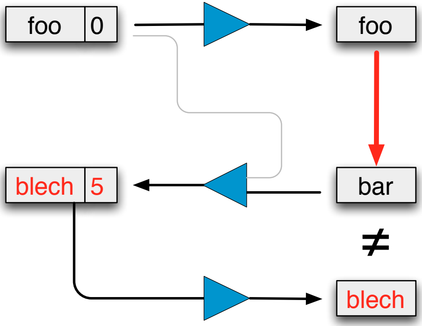
About Preservation of Changes



About Preservation of Changes



About Preservation of Changes

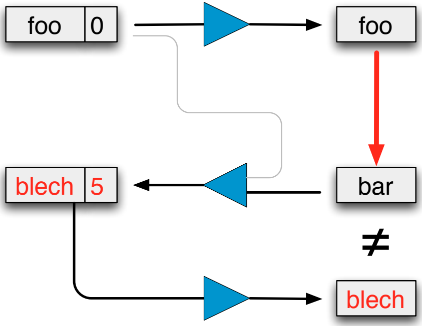


Principle: Updates should be translated exactly.

To prevent this, the PutGet law:

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

About Preservation of Changes



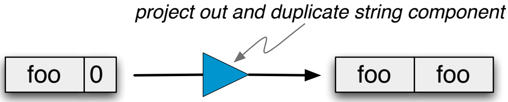
Principle: Updates should be translated exactly.

To prevent this, the PutGet law:

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

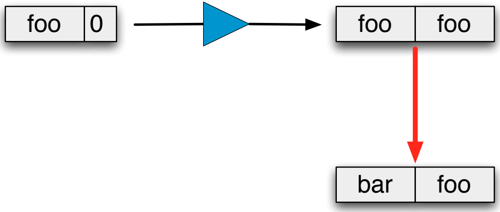
NB: For this, `put` s must be injective for every s.

Somewhat more Challenging

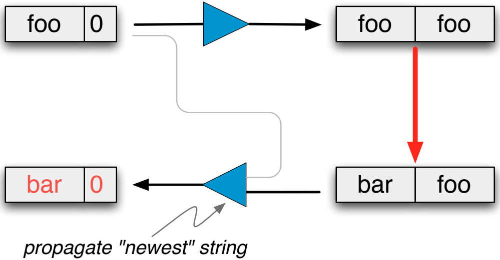


project out and duplicate string component

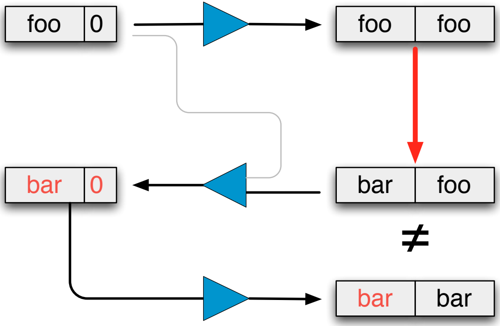
Somewhat more Challenging



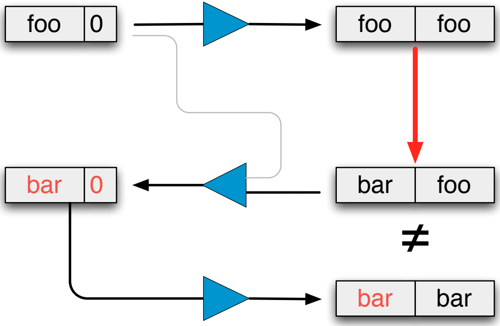
Somewhat more Challenging



Somewhat more Challenging

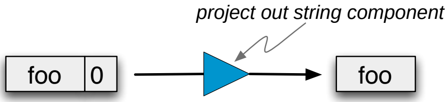


Somewhat more Challenging

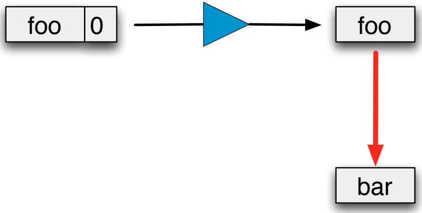


If we want to allow such behavior, we need to weaken the PutGet law (and people have done so).

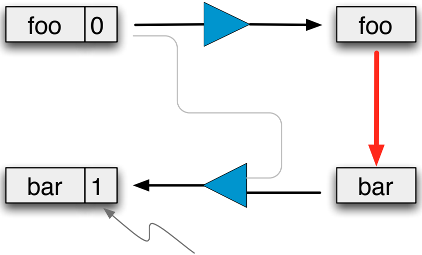
About Consistent Composition



About Consistent Composition

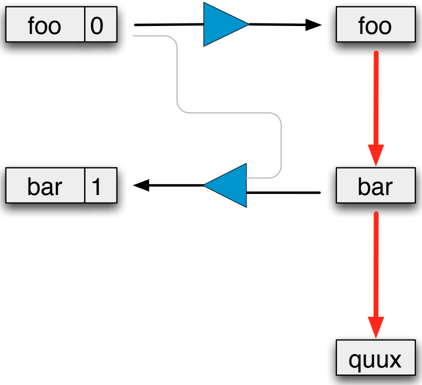


About Consistent Composition

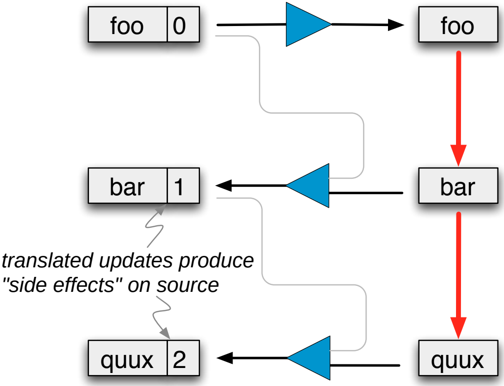


increment numeric component
if string component has changed

About Consistent Composition

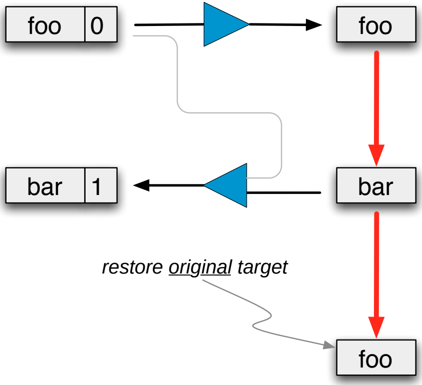


About Consistent Composition



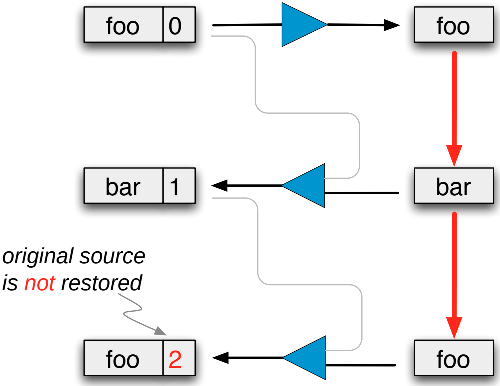
translated updates produce "side effects" on source

About Consistent Composition



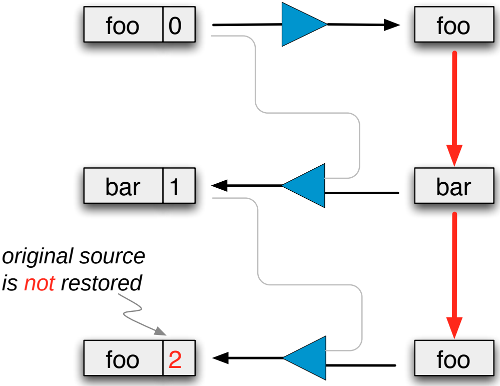
restore original target

About Consistent Composition



original source
is not restored

About Consistent Composition



To prevent this, the PutPut law:

$$\mathtt{put}\ (\mathtt{put}\ s\ v)\ v' = \mathtt{put}\ s\ v'$$

Less Debatable

Actually a consequence of GetPut and PutGet, the PutTwice law:

$$\texttt{put}\ (\texttt{put}\ s\ v)\ v = \texttt{put}\ s\ v$$

Less Debatable

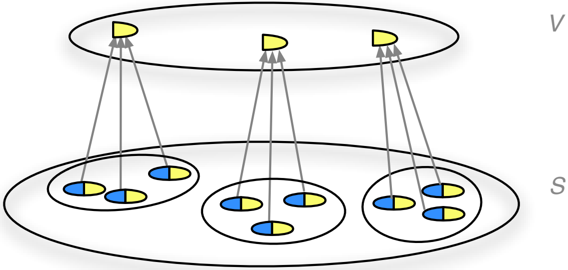
Actually a consequence of GetPut and PutGet, the PutTwice law:

$$\texttt{put}\ (\texttt{put}\ s\ v)\ v = \texttt{put}\ s\ v$$

We'll get back to this property in a moment.

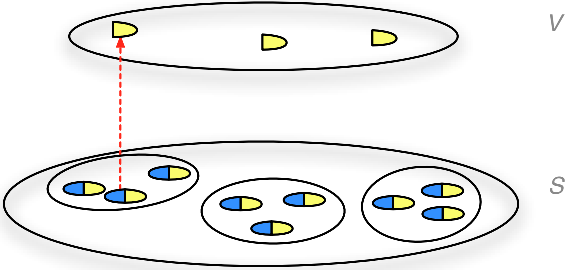
Ambiguity of `put`

How many `puts` are there?



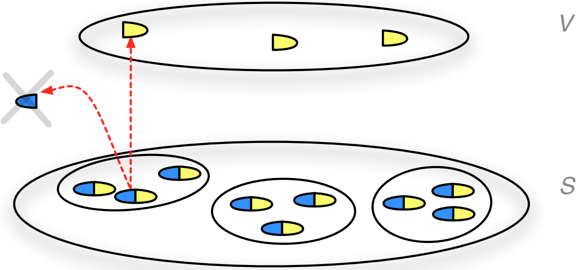
Due to non-injectivity, `get` can map many objects from S onto the same object from V.

How many `puts` are there?



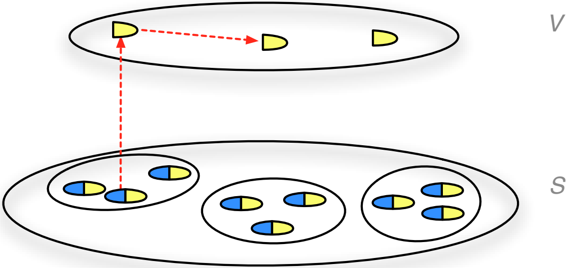
In essence, `get` projects out part of the information in the source object…

How many `puts` are there?



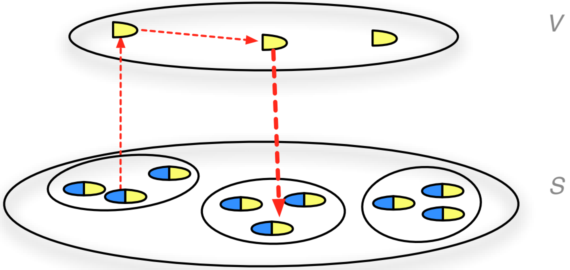
In essence, `get` projects out part of the information in the source object…and throws away the rest.

How many `puts` are there?



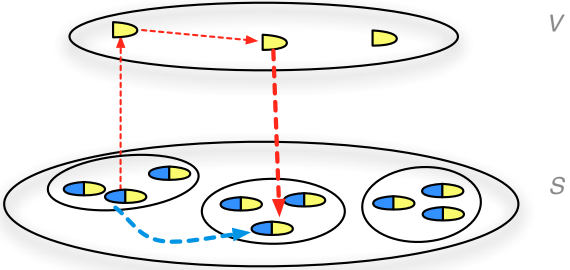
After an update,

How many `puts` are there?



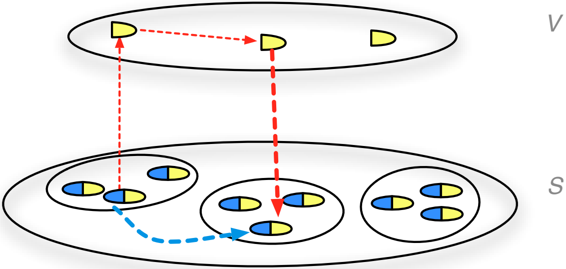
After an update, the “view part” of the new source object is fixed by PutGet. . .

How many `puts` are there?



After an update, the “view part” of the new source object is fixed by PutGet… and if the lens obeys PutPut, the “projected away part” is fixed to be exactly the one from the original source.

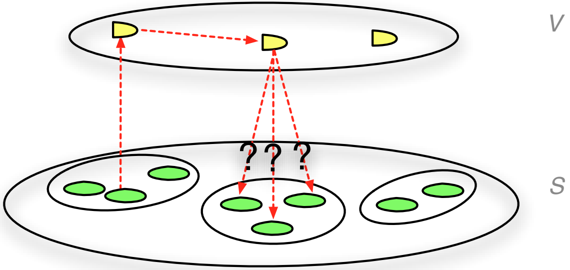
How many `puts` are there?



After a … source
object … beys
PutPut … be
exactly the one from the original source.

Even this doesn’t mean that there is only exactly one “very well-behaved” `put` per `get`!

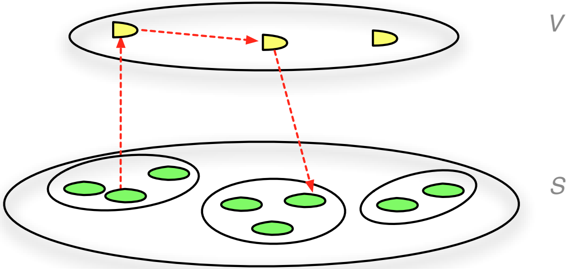
How many `puts` are there?



Moreover, if the lens *doesn't* need to obey PutPut, then the behavior of `put` is much less constrained…

…and there are even more `put`s to choose from!

How many `puts` are there?



Moreover, if the lens *doesn't* need to obey PutPut, then the behavior of `put` is much less constrained…

…and there are even more `put`s to choose from!

So, definitely need extra information to select one.

On the Other Hand…

… there is only one `get` per "well-behaving" `put`!

On the Other Hand…

… there is only one `get` per “well-behaving” `put`!

Specifically, if `put` is surjective, is injective for every s, and satisfies PutTwice, then there is *exactly* one `get` such that the two together satisfy GetPut and PutGet.

On the Other Hand...

... there is only one `get` per “well-behaving” `put`!

Specifically, if `put` is surjective, is injective for every *s*, and satisfies PutTwice, then there is *exactly* one `get` such that the two together satisfy GetPut and PutGet. And, there are equivalent, even nicer conditions formulated just in terms of `put` as well.

[Fischer, Hu, Pacheco]

On the Other Hand...

... there is only one `get` per “well-behaving” `put`!

Specifically, if `put` is surjective, is injective for every *s*, and satisfies PutTwice, then there is *exactly* one `get` such that the two together satisfy GetPut and PutGet. And, there are equivalent, even nicer conditions formulated just in terms of `put` as well.
[Fischer, Hu, Pacheco]

There are even first concrete bidirectionalization techniques derived from this `put`-based approach!

Conclusion / Discussion (?)

“Solved”

- a lot of very nice definitive work on semantics
- successful methods for automatic derivation of reasonable `put`- from `get`-functions on strings, trees, and graphs (?)
  - combinator languages with powerful type systems
  - program transformations based on constant-complement
  - query languages with automatic tracing
  - grammar-based approaches

Open Problems

Leaving the academic niche:

- “How to deliver BX to the masses? Some effective way to integrate BX with existing general programming languages would be nice. Most tools/languages are very academic, and I don’t see them being used for industrial case studies. . . ”
- “But I think to really achieve world domination, a BX framework will need to make substantial progress on having an attractive and intuitive front-end.”

Open Problems

Tackling ambiguities effectively:

- “Can we design a declarative language that can be used to describe any intentional bidirectional behavior (i.e., have full control of bidirectional behavior)?”
- “We still lack effective, intuitive (user-friendly) and generic mechanisms to tame the non-determinism of backwards transformation.”
- “Ability to control the choice between multiple valid backward transformation results.
  [. . .] clarify to what extent user can control by writing different get (forward) transformations.”

Open Problems

Handling richer semantic domains:

- "[. . .] still no effective solution for non-tree shaped domains."
- "Bx on ordered graphs (outgoing edges are ordered) and graphs in which ordered and unordered edges are mixed."
- "Handling of constraints over the domains (that is, handling non CFG-like domains). DB people have some work on this (handling keys, functional dependencies, inclusion dependencies, etc), but the issue seems ignored by PL people."

“Conclusion”

There is a lot of potential and possible inspiration from PL land for the general area of BX.

Challenges remain:

- scaling up in every way
- providing control over nondeterminism
- capturing user/programmer intentions
- handling richer structures/domains
- running efficiently