Bidirectionalization for Free with Runtime Recording

Or, a Light-Weight Approach to the View-Update Problem

Kazutaka Matsuda*
Meng Wang**

* University of Tokyo
** Chalmers University of Technology

PPDP 2013 @ Madrid, Spain

This Talk is About ...

- A lightweight & automatic way to construct ***bidirectional transformation*** from unidirectional transformation

Bidirectional Transformation

- = transformation pair: **`get`** & **`put`**
  - formalization of the view-update problem

[Bansilhon&Spyratos81, Foster+07, ...]

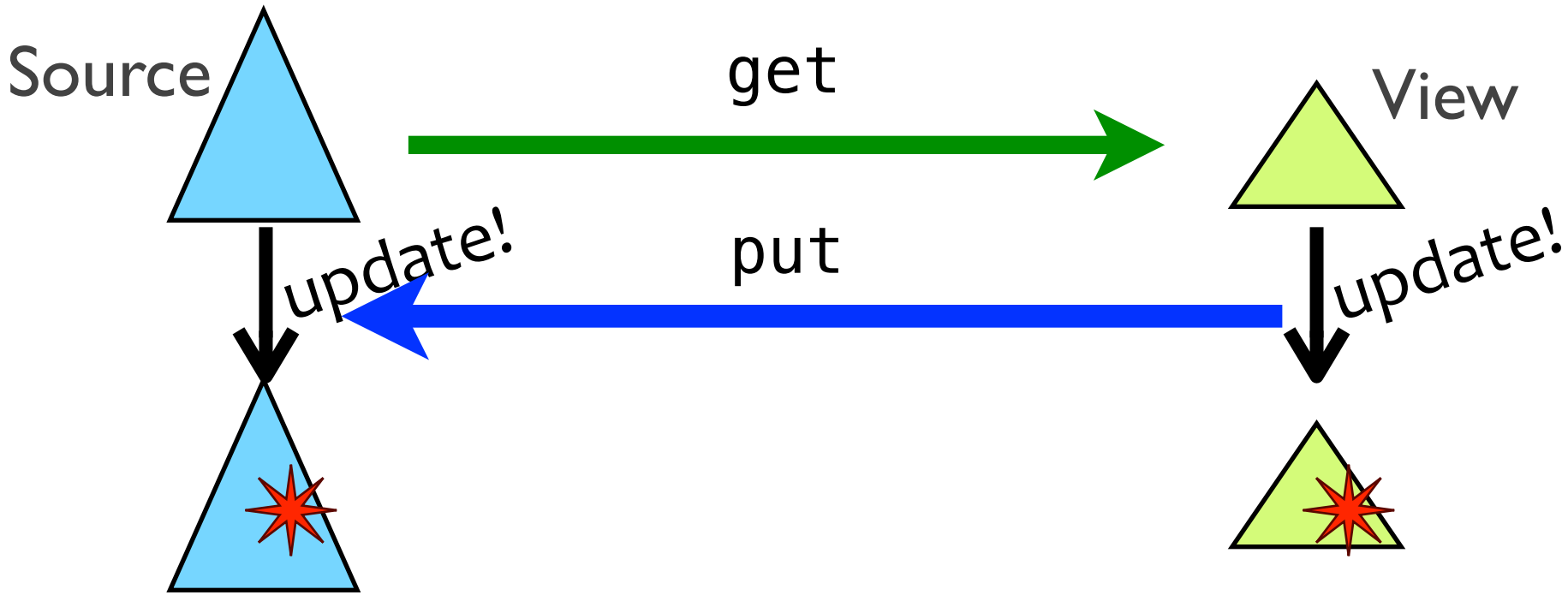
Bidirectional Transformation

- = transformation pair: **`get`** & **`put`**
  - formalization of the view-update problem

[Bansilhon&Spyratos81, Foster+07, ...]

Bidirectional Transformation

- = transformation pair: **`get`** & **`put`**
  - formalization of the view-update problem



[Bansilhon&Spyratos81, Foster+07, ...]

Well-Behavedness

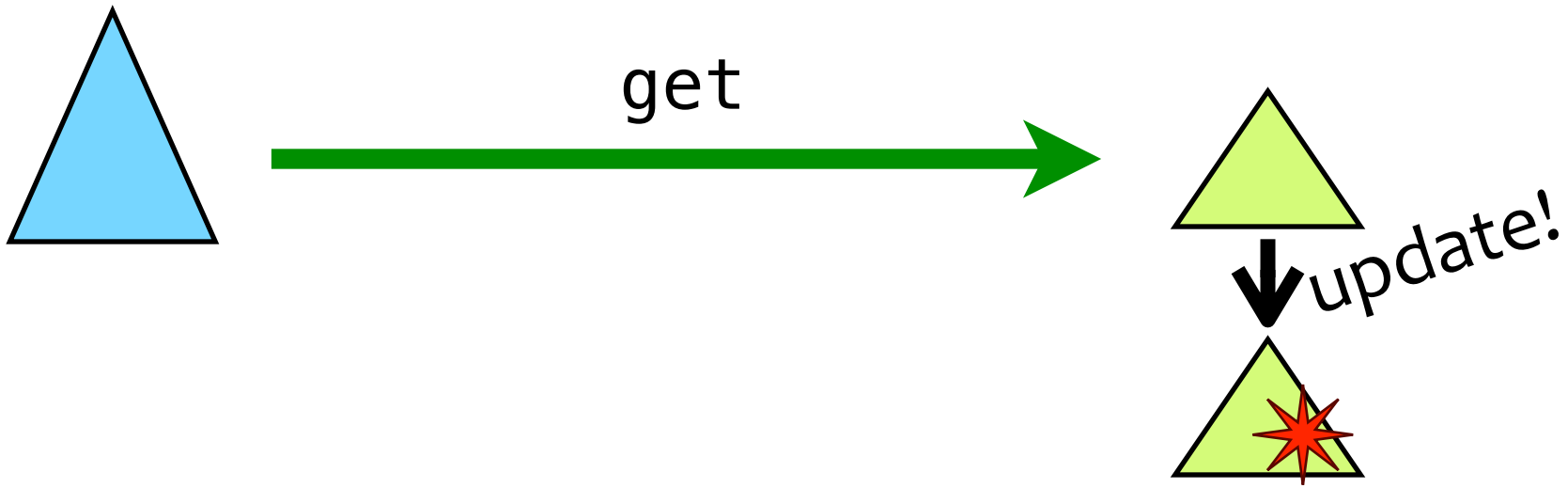
- ***Roundtrip Law***
- ...

[Bansilhon&Spyratos81, Foster+07, ...]

Roundtrip Law

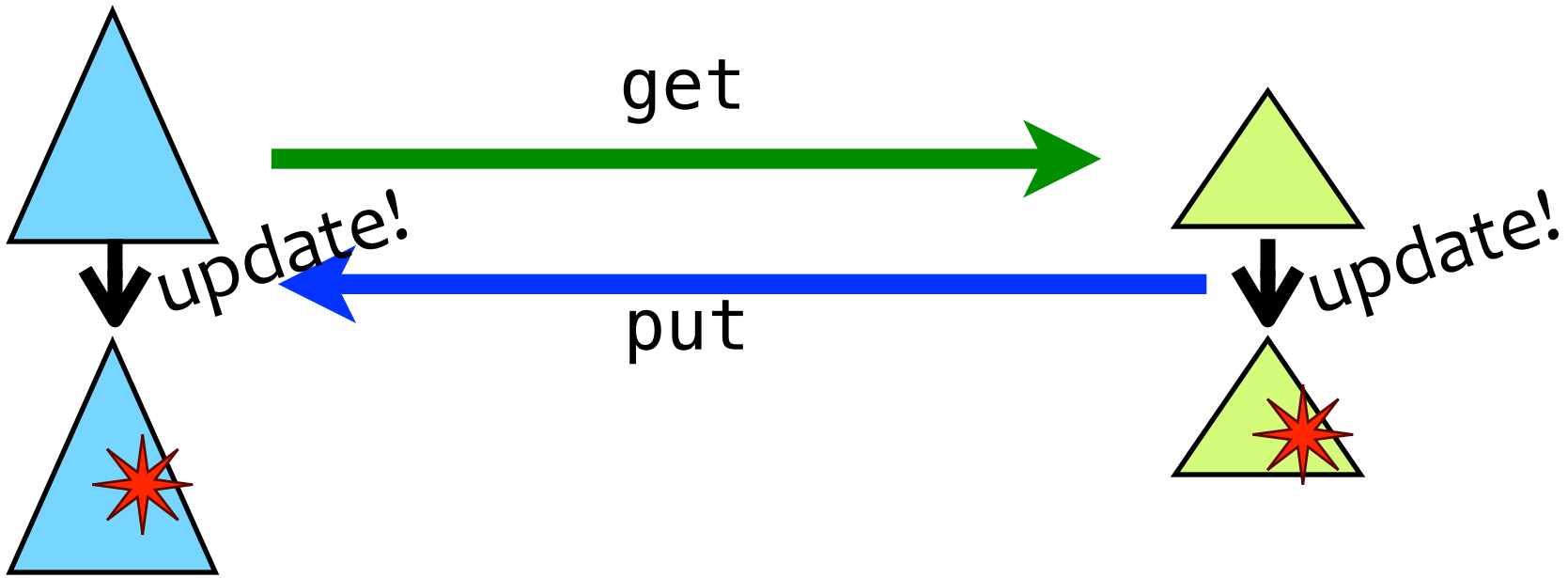
[Bansilhon&Spyratos81, Foster+07, ...]

Roundtrip Law



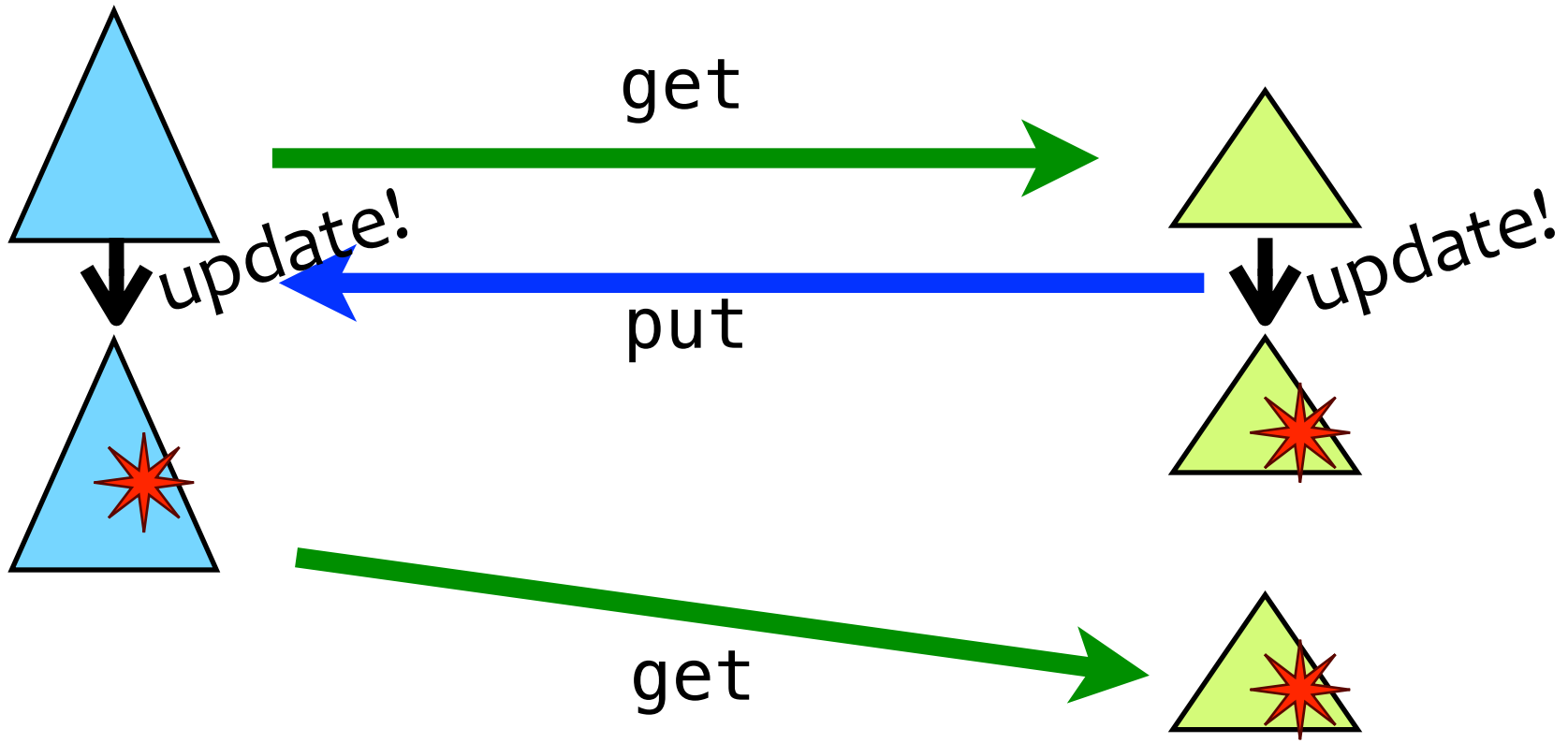
[Bansilhon&Spyratos81, Foster+07, ...]

Roundtrip Law



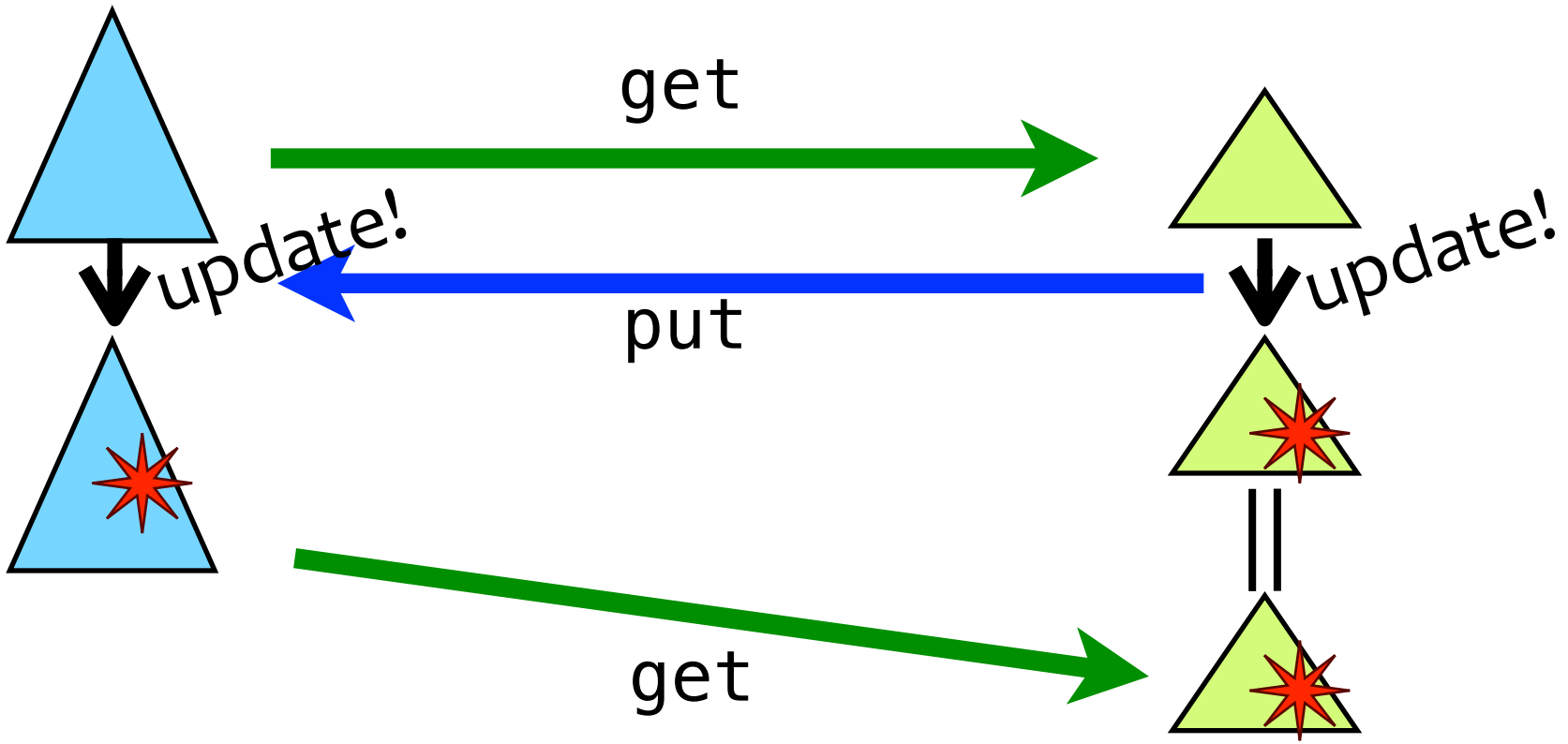
[Bansilhon&Spyratos81, Foster+07, ...]

Roundtrip Law



[Bansilhon&Spyratos81, Foster+07, ...]

Roundtrip Law



[Bansilhon&Spyratos81, Foster+07, ...]

Goal

- To construct well-behaved bidirectional transformation

Existing Approach

Bidirectionalization for Free

Result [Voigtländer 09]

We can derive a well-behaved bidirectional transformation from a get-function of type

$$\forall a.\ T_1\ a \to T_2\ a$$

polymorphic

- Advantage: ***user-friendliness***
  - We can use ***arbitrary*** languages
    - as long as the language respects parametricity [Reynolds83]

Limitation

- The polymorphism requirement is too restrictive

Construction of new elements

```
<bib>
 {
  for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

Comparison with constants

from XML Query Use Cases
http://www.w3.org/TR/xquery-use-cases/#xmp-queries-results-q1

Our Contribution

- Extends "bidirectionalization for free" [Voigtländer 09]
  - to support many XML transformations
    - that may
      - construct new elements
      - compare elements with constants

Key techniques:

- a ***type class*** to abstract these operations
- ***runtime recording*** to guarantee well-behavedness

Outline

- Review: "bidirectionalization for free" [Voigtländer 09]
- Our Approach

Bidirectionalization for Free

Result [Voigtländer 09]

We can derive a well-behaved bidirectional transformation from a get-function of type

$$\forall a.\ T_1\ a \to T_2\ a$$

Bidirectionalization for Free

Result [Voigtländer 09]

We can derive a well-behaved bidirectional transformation from a get-function of type

```
∀a. [a] → [a]
```

Idea

Result [Voigtländer 09]

We can derive a well-behaved bidirectional transformation from a get-function of type

```
∀a. [a] → [a]
```

- Idea: to exploit

  "a get-function is ***polymorphic***"
  - free theorem [Reynolds83, Wadler89]
    - theorem obtained only by "being polymorphic"

Prop. of Polymorphic Func.

Quiz

What are properties of `f :: ∀a. [a] → [a]`

Answers

- All the elems in the output come from the input
- Only the "shape" determines the behavior

Example: `f = tail`

```
f [1,2]         = [2]
f [1,2,3]       = [2,3]
f ["A","B"]     = ["B"]
f ["A","B","C"] = ["B","C"]
```

Prop. of Polymorphic Func.

Quiz

What are properties of `f :: ∀a. [a] → [a]`

Answers

- All the elems in the output come from the input
- Only the "shape" determines the behavior

Free theorem is a generalization of them

Example: `f = t`

```
f [1,2]         = [2]
f [1,2,3]       = [2,3]
f ["A","B"]     = ["B"]
f ["A","B","C"] = ["B","C"]
```

Idea to Construct "Put"

```
get :: ∀a.[a]→[a]
```

Given

```
[A,B,B]
```

Idea to Construct "Put"

```
get :: ∀a.[a]→[a]
```

Given

```
[A,B,B]
```

Idea to Construct "Put"

```
get :: ∀a.[a]→[a]
```

Given

Construct "Put"


```
get :: ∀a.[a]→[a]
```

Given

```
[A,B,B]  ──get──▶  [B,B]
```

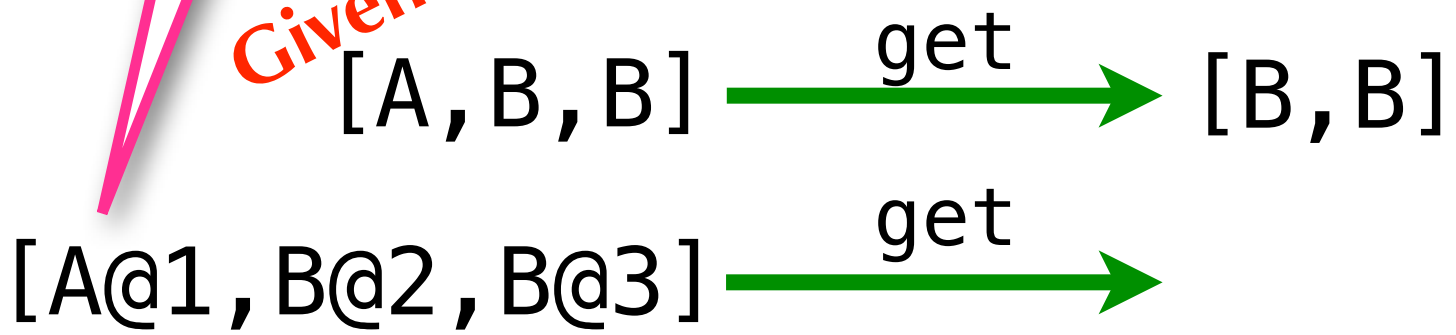
```
[A@1,B@2,B@3]
```

Idea to Construct "Put"

Assign locations

```
get :: ∀a.[a]→[a]
```

Given

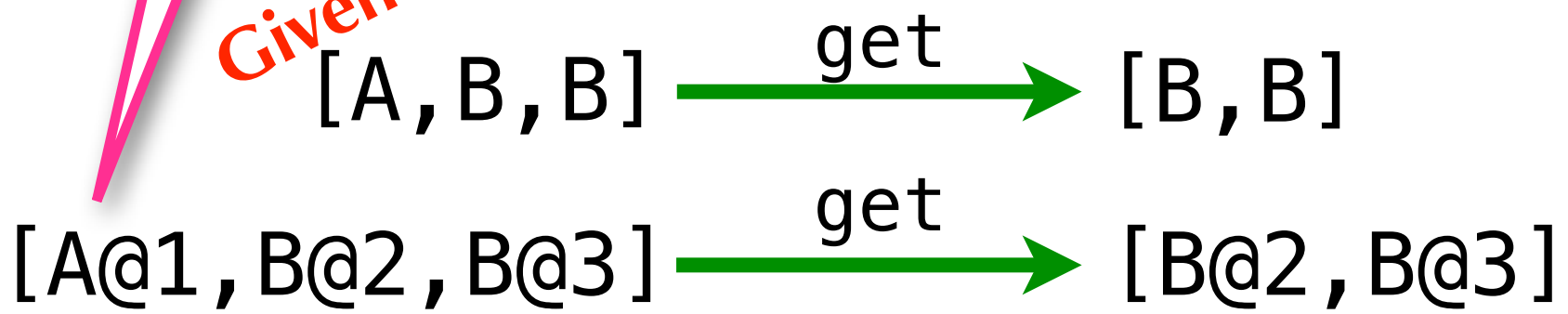


Idea to Construct "Put"

Assign locations

```
get :: ∀a.[a]→[a]
```

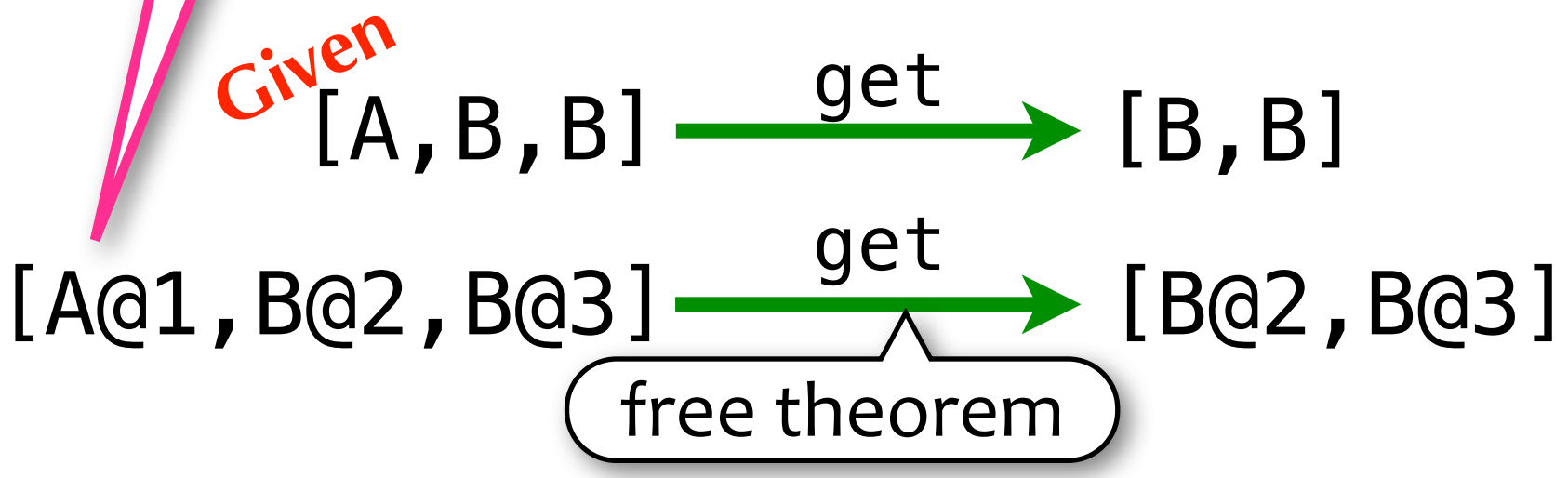
Given



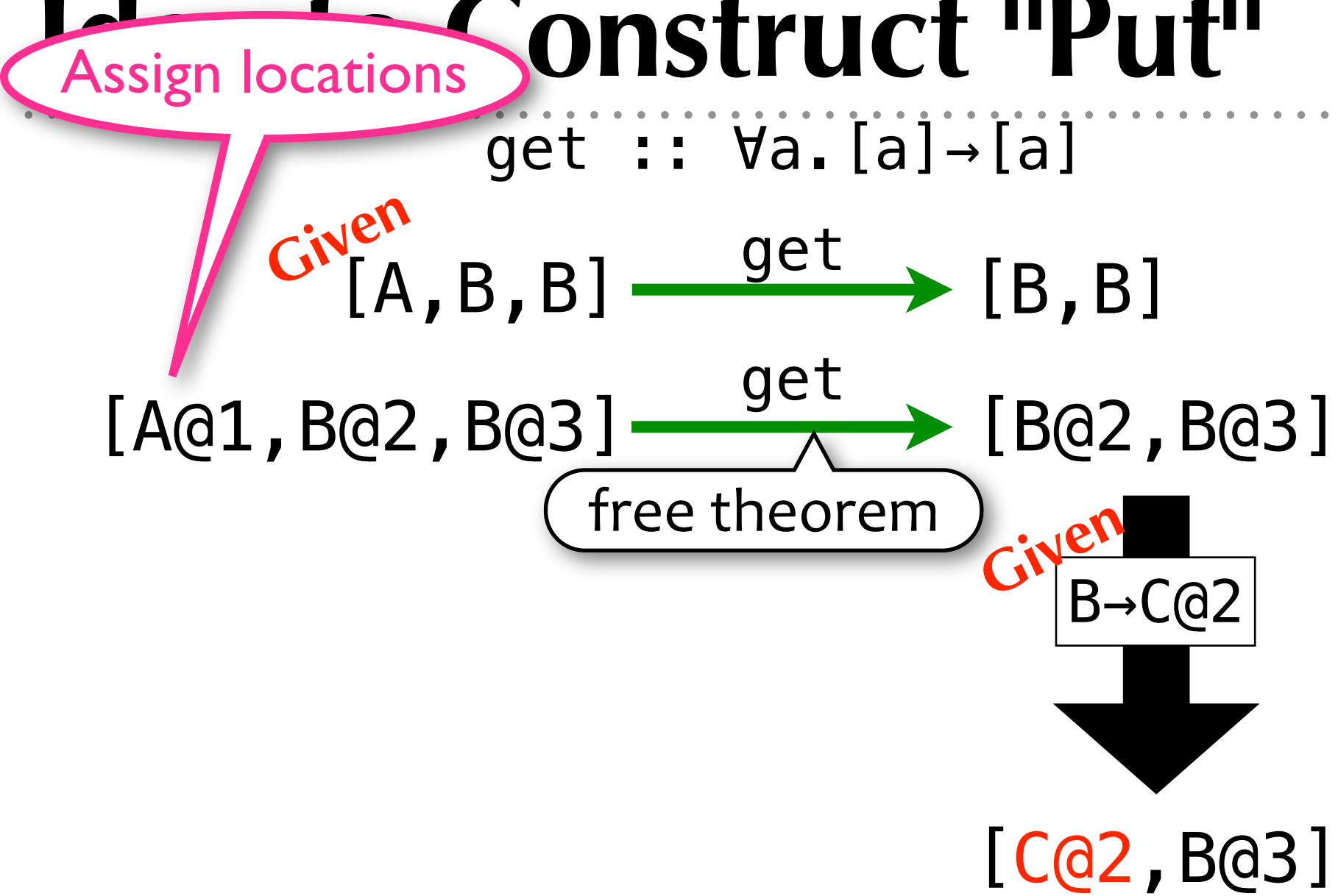
Idea to Construct "Put"

Assign locations

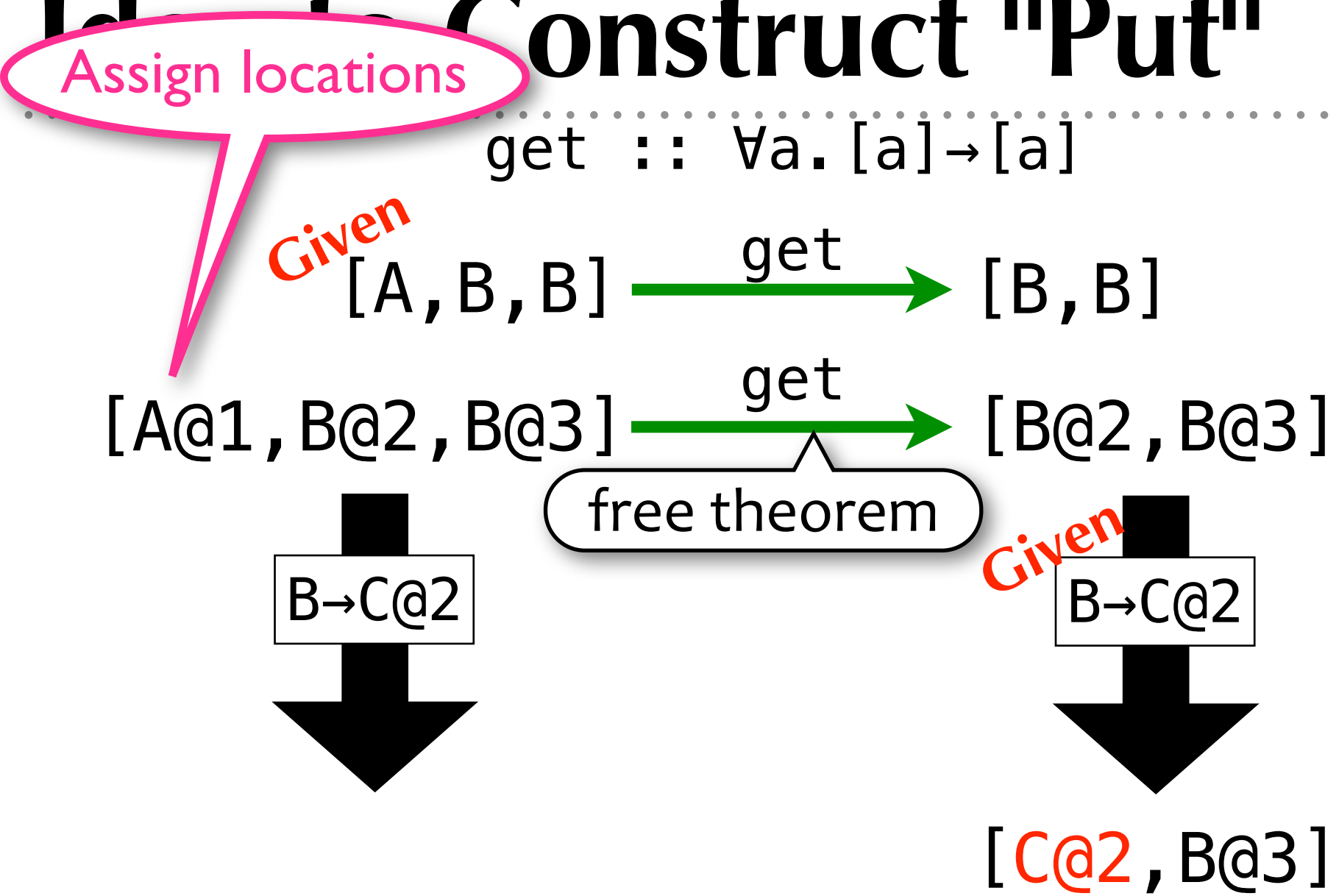
```
get :: ∀a.[a]→[a]
```



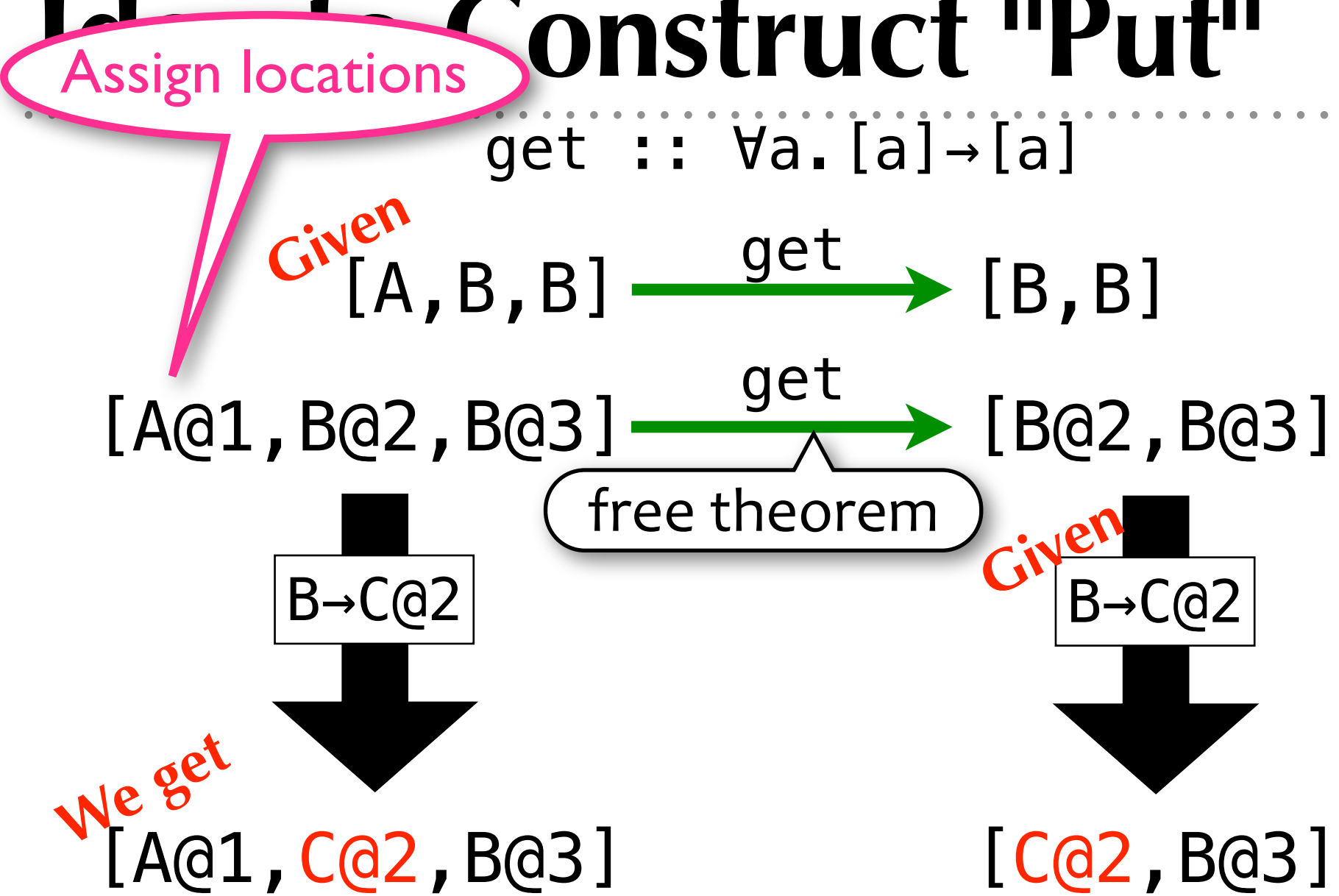
Idea to Construct "Put"



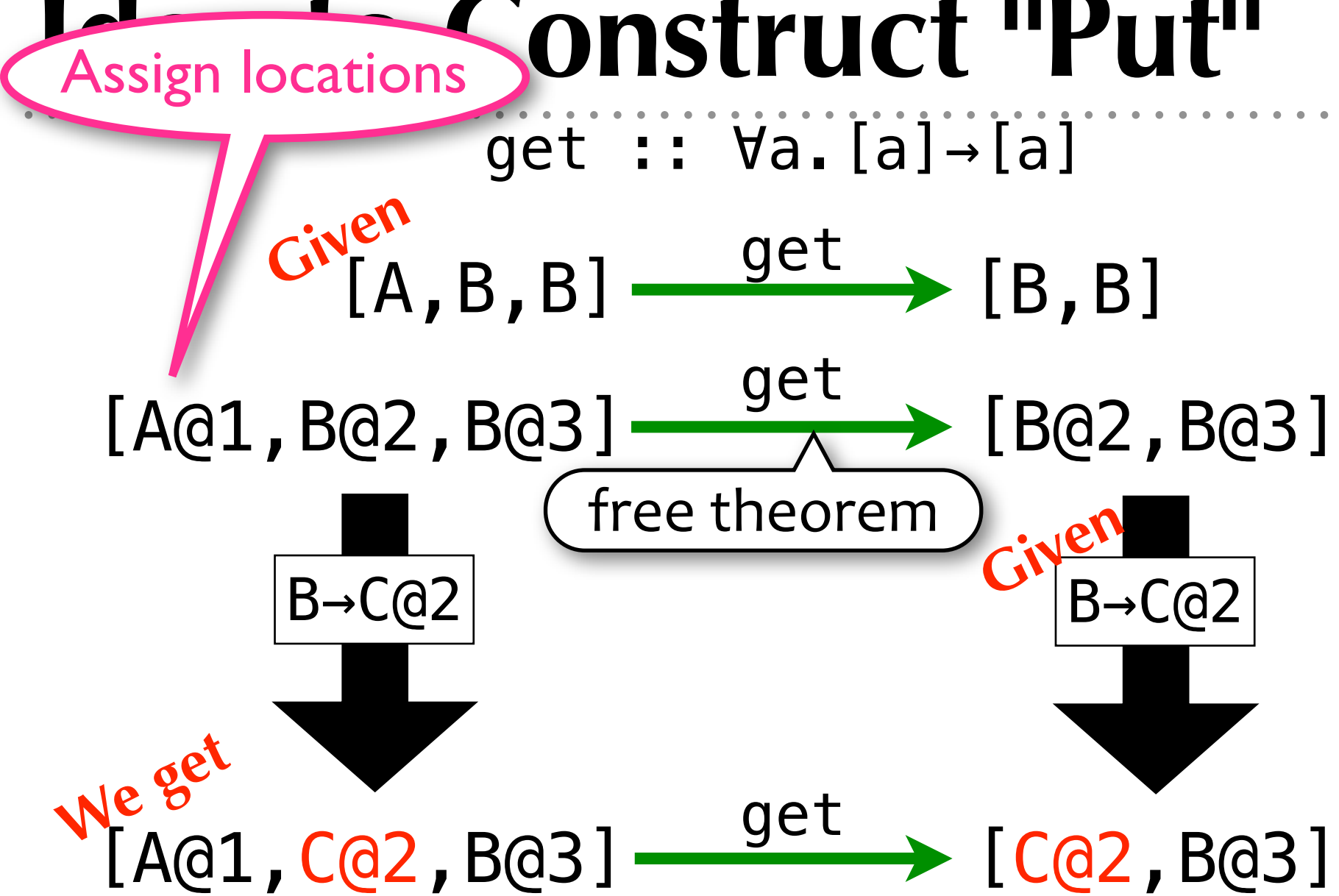
Idea to Construct "Put"



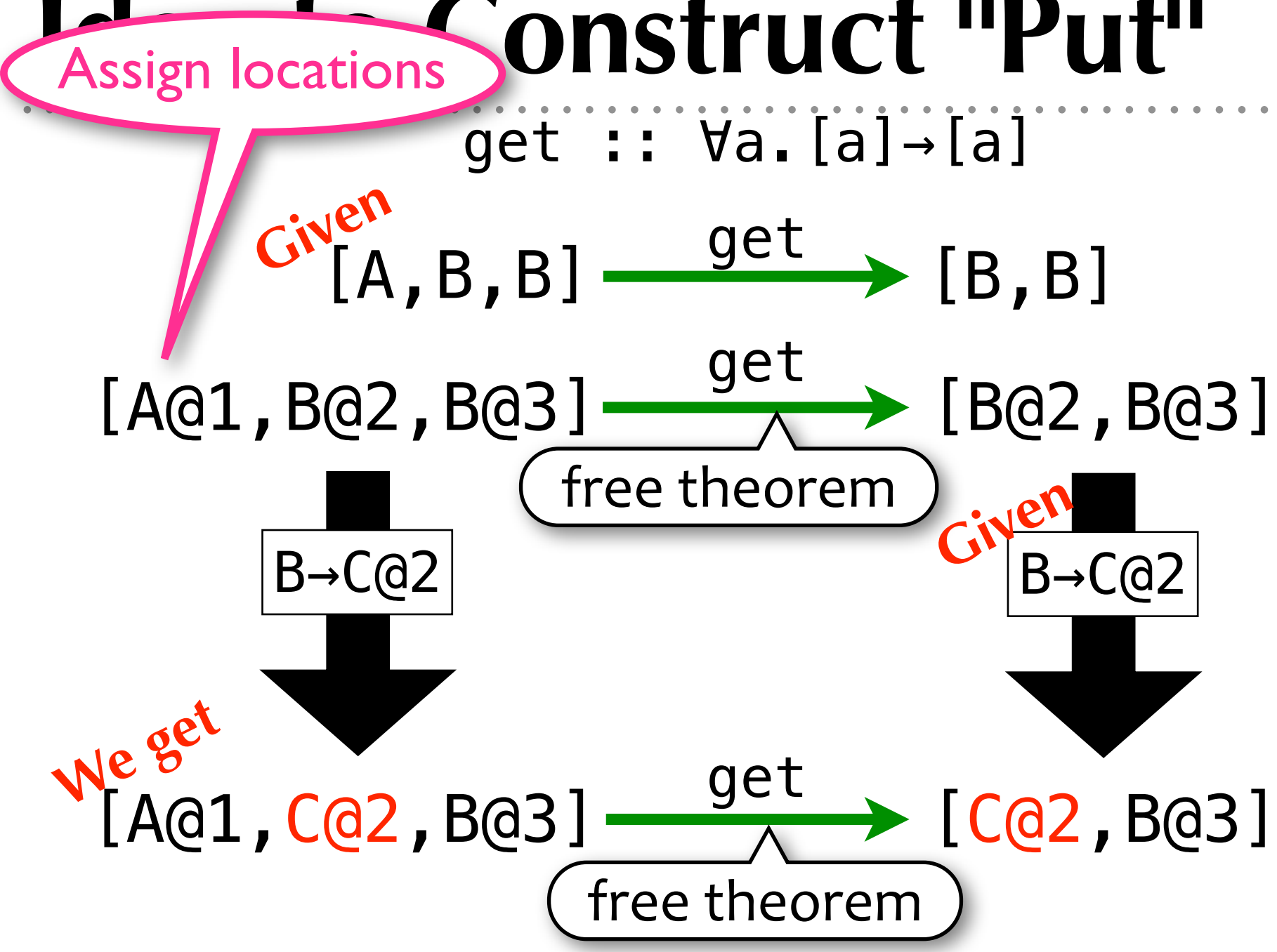
Idea to Construct "Put"



Idea to Construct "Put"



Idea to Construct "Put"



Limitation (Repeated)

- The polymorphism requirement is too restrictive

Construction of new elements

```
<bib>
 {
  for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

Comparison with constants

from XML Query Use Cases
http://www.w3.org/TR/xquery-use-cases/#xmp-queries-results-q1

Outline

- Review: "bidirectionalization for free" [Voigtländer 09]
- **Our Approach**

Simplified Example

- `findA`:
  - Extracts the first sublist starting "A", prepending "R"

Naive Realization

- Monomorphic, impossible to apply "bidirectionalization for free"

```
findA :: [String] → [String]
findA x = "R":go x
   where
     go [] = []
     go (a:x) = if a == "A" then
                    a:x
                else
                    go x
```

First Trial

- A type class to abstract the operations

```
findA :: ∀a. StringLike a => [a] → [a]
findA x = new "R":go x
   where
     go [] = []
     go (a:x) = if a `eq` new "A" then
                    a:x
                else
                    go x
```

```
class StringLike a where
  new :: String → a
  eq  :: a → a → Bool
```

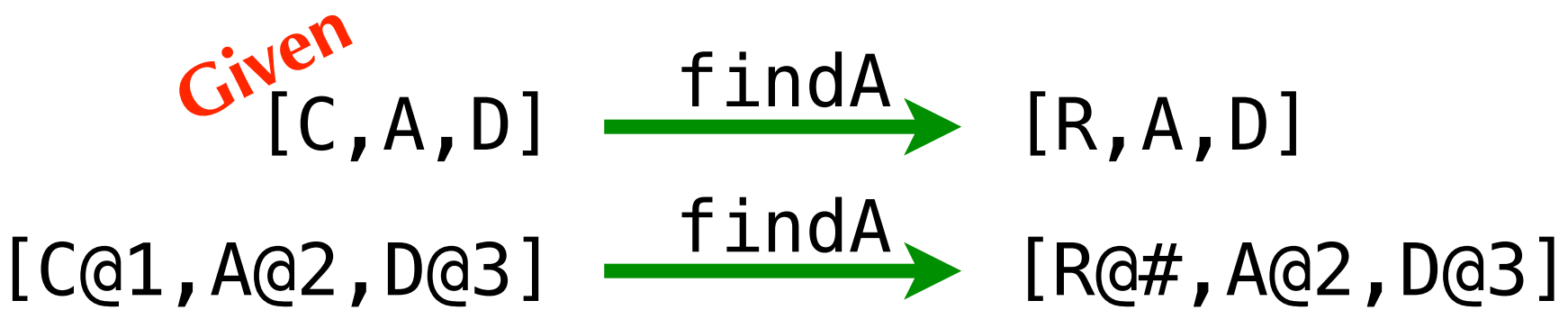
Abstracting:
- new data creation
- equivalence-check

Successful Case

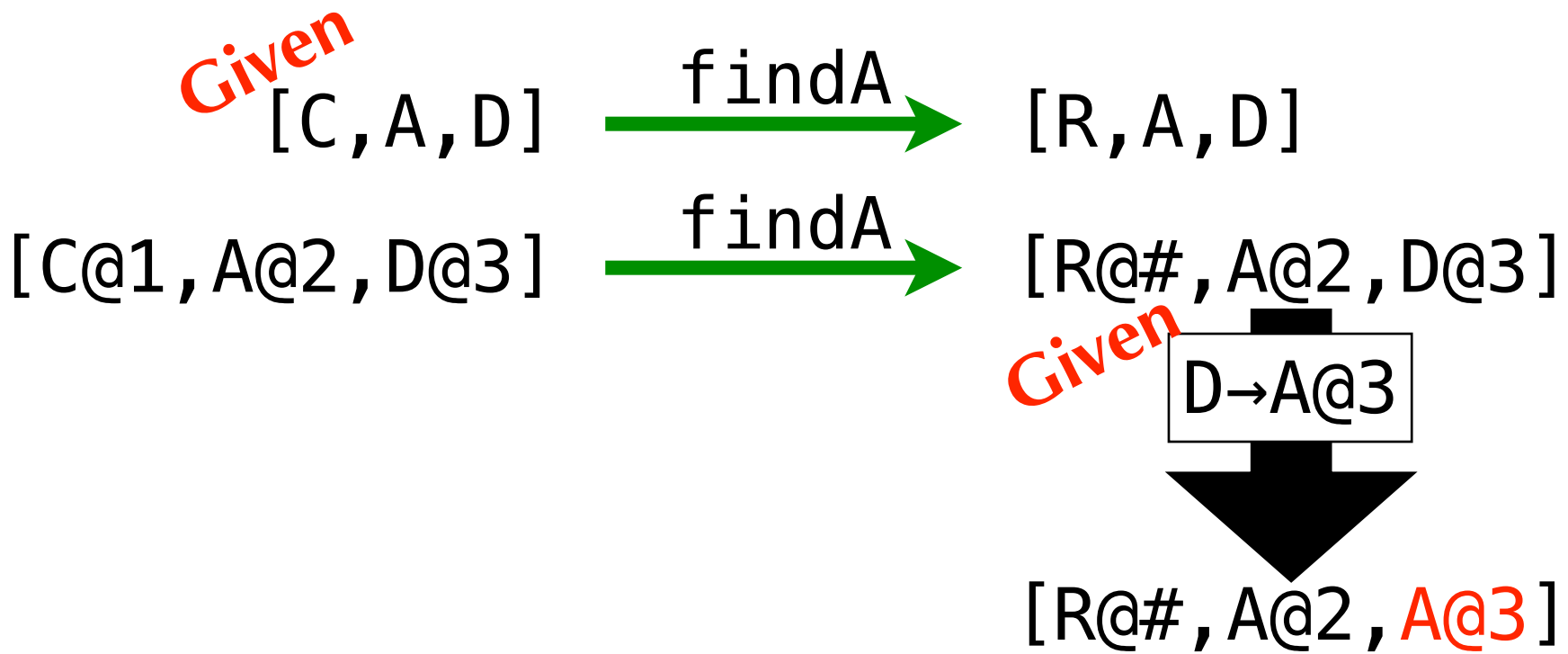
Successful Case


```
[C@1,A@2,D@3]
```

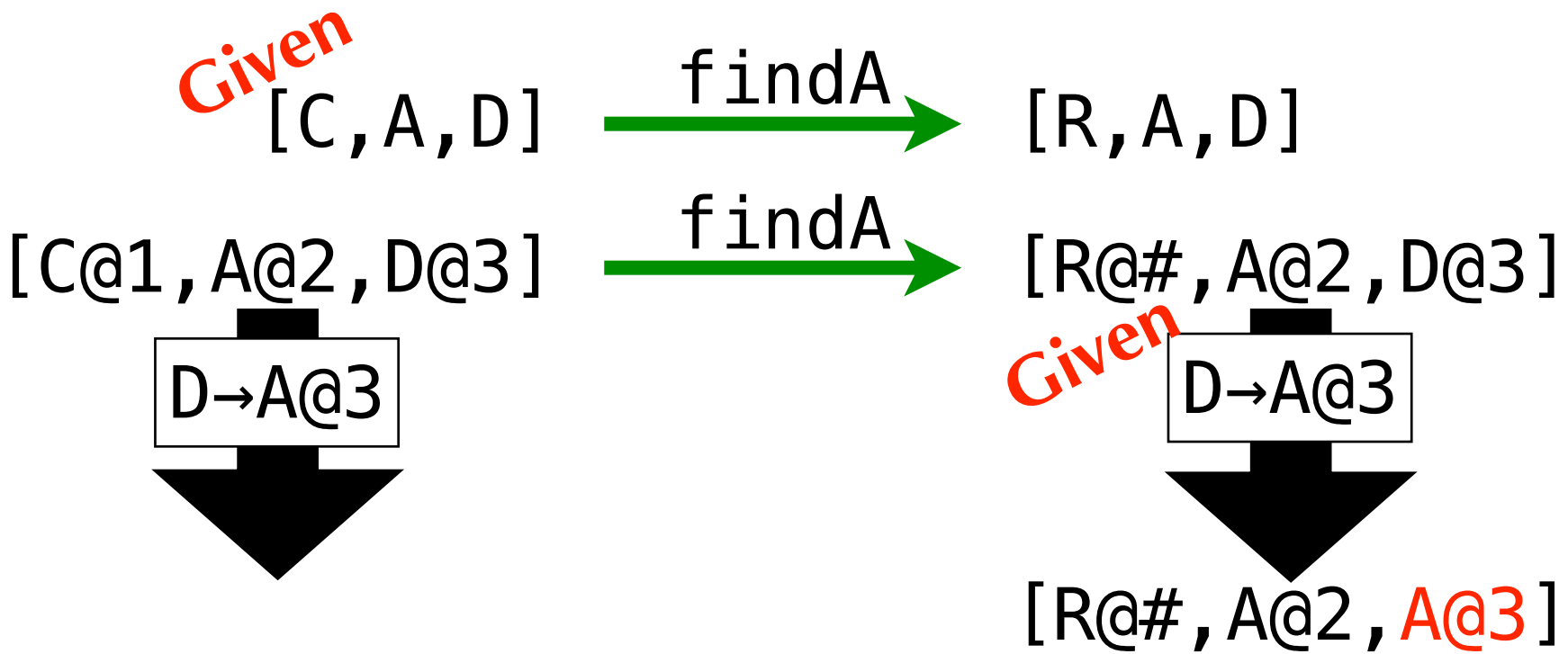
Successful Case



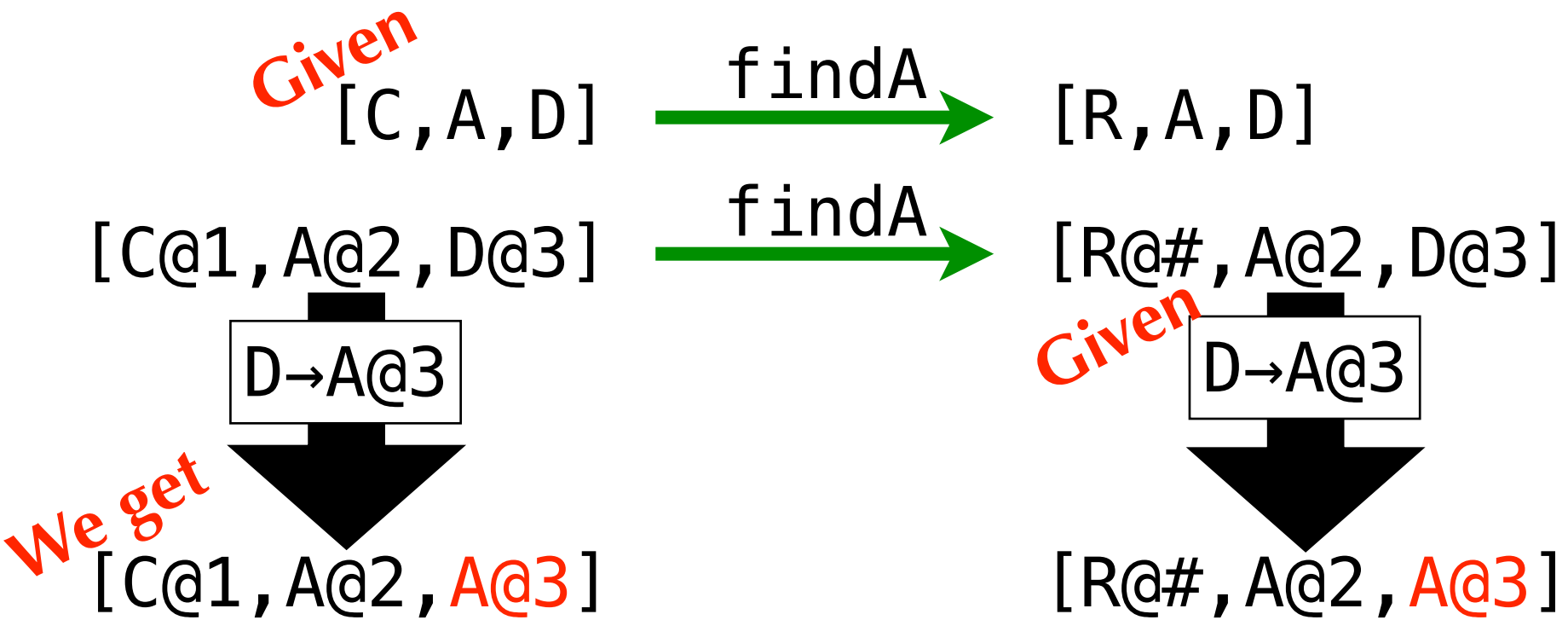
Successful Case



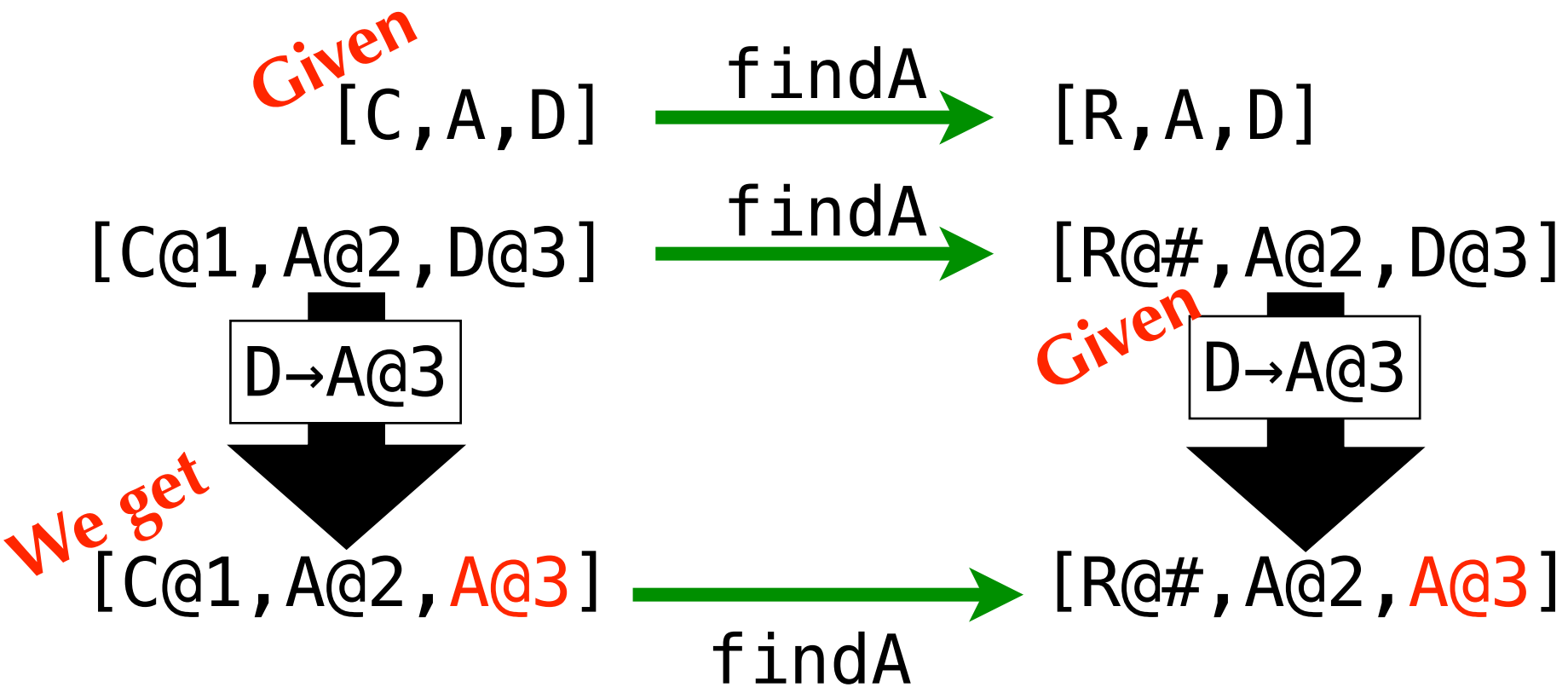
Successful Case



Successful Case

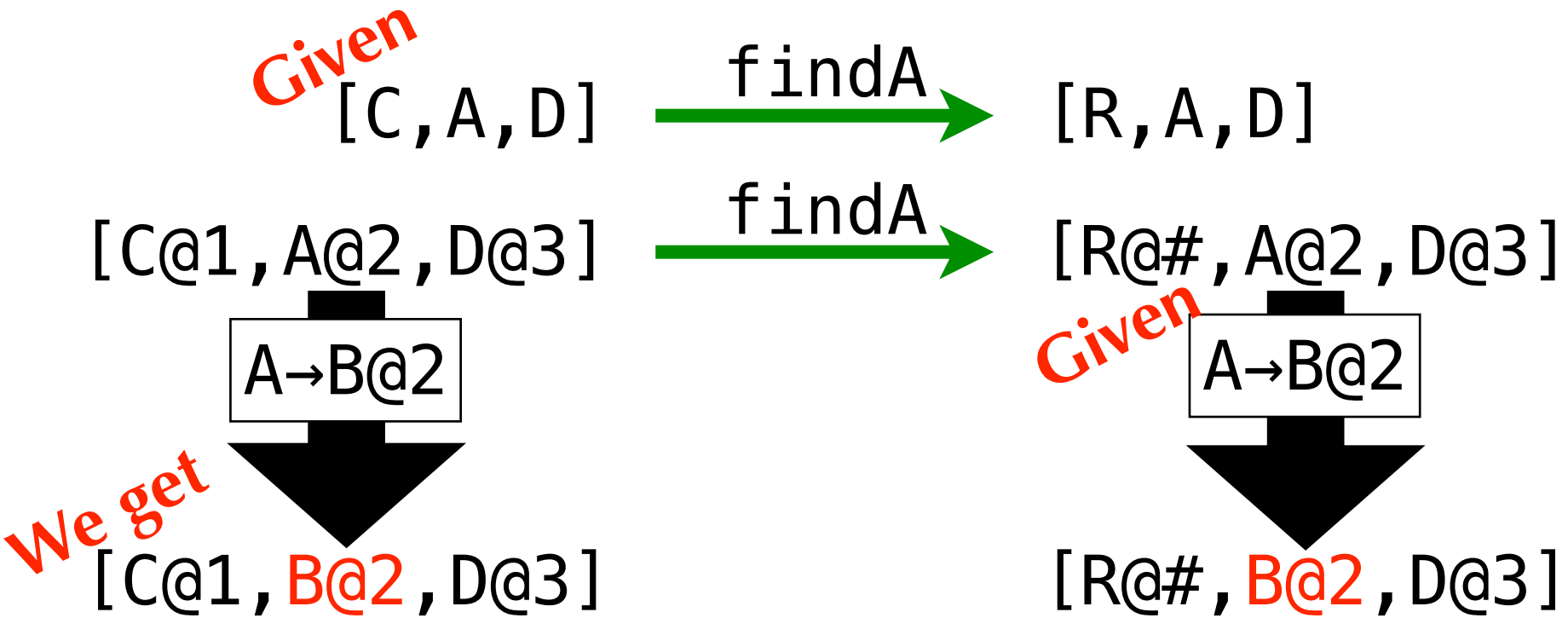


Successful Case



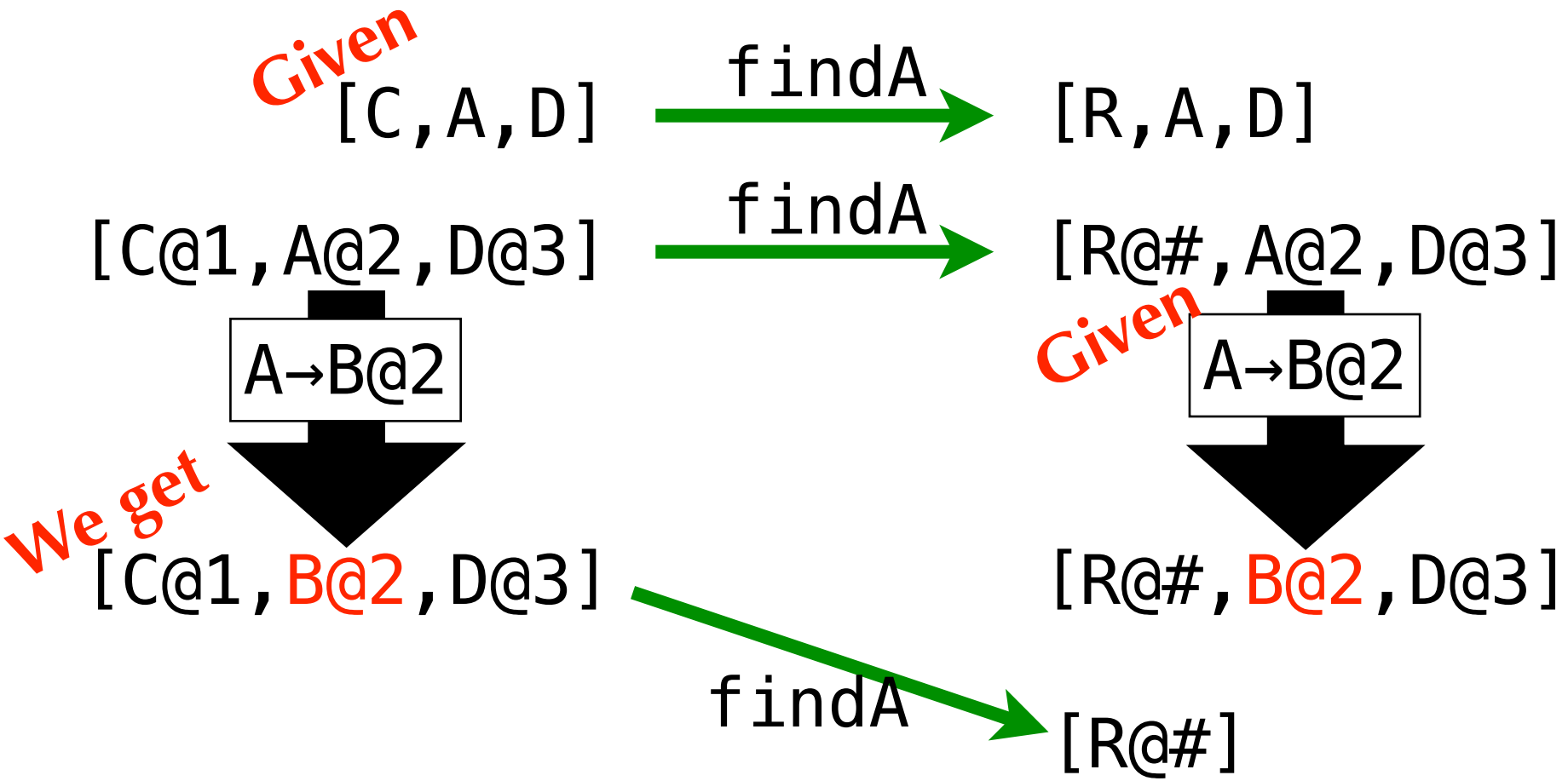
Problem

- Well-behavedness is violated



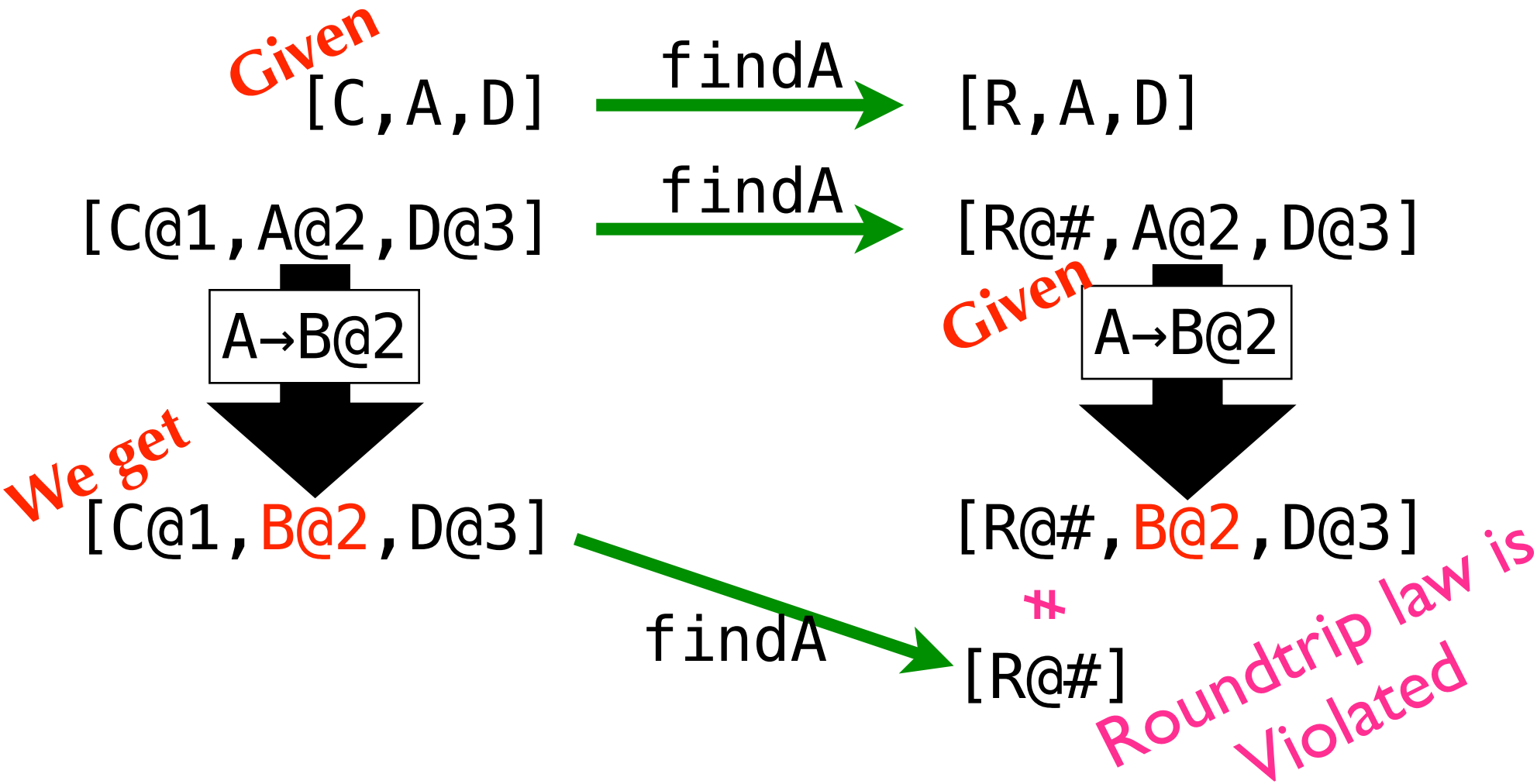
Problem

- Well-behavedness is violated



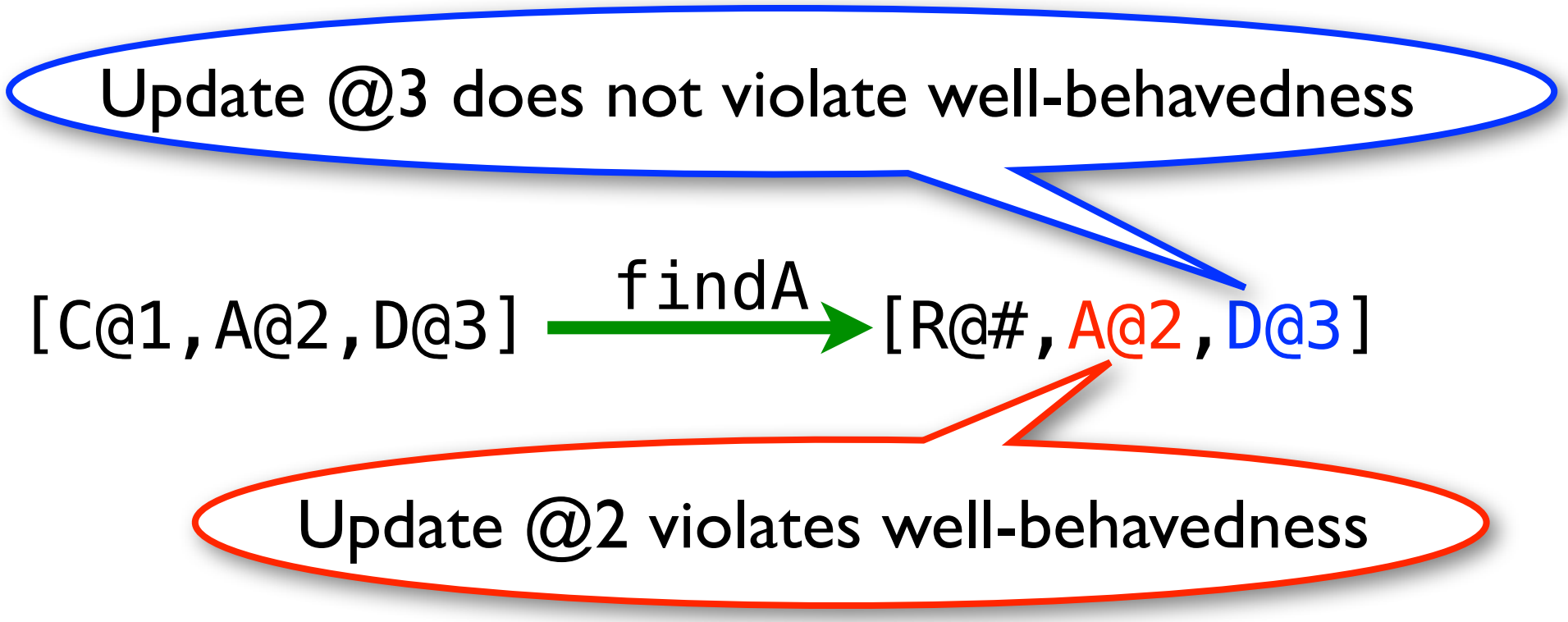
Problem

- Well-behavedness is violated



Question

What causes the difference?



Answer

A@2 is ***observed*** by `findA`

but D@3 is not


```
findA :: ∀a. StringLike a => [a] → [a]
findA x = new "R":go x
   where
     go [] = []
     go (a:x) = if a `eq` new "A" then
                   a:x
                else
                   go x
```

Changing A@2 affects control path

Our Idea

- To use a ***monad*** to record ***observations***
- To check if the recorded ***observations*** are kept consistent through updates

```
findA :: ∀a.∀m.StringLike a m => [a] → m [a]
…
```

```
class Monad m => StringLike a m where
   new :: String → a
   eq  :: a → a → m Bool
```

Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```

Given

```
[C,A,D]
```

Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```

Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```

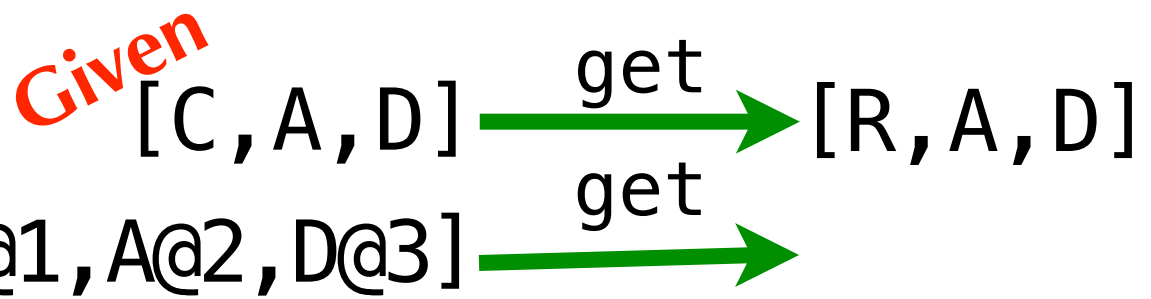
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```


```
[C@1,A@2,D@3]
```

Our Bidirectionalization

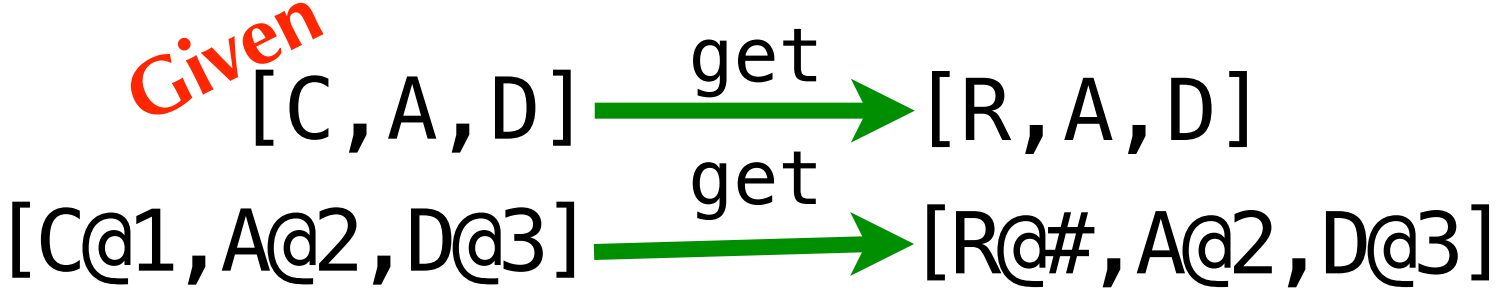
```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```

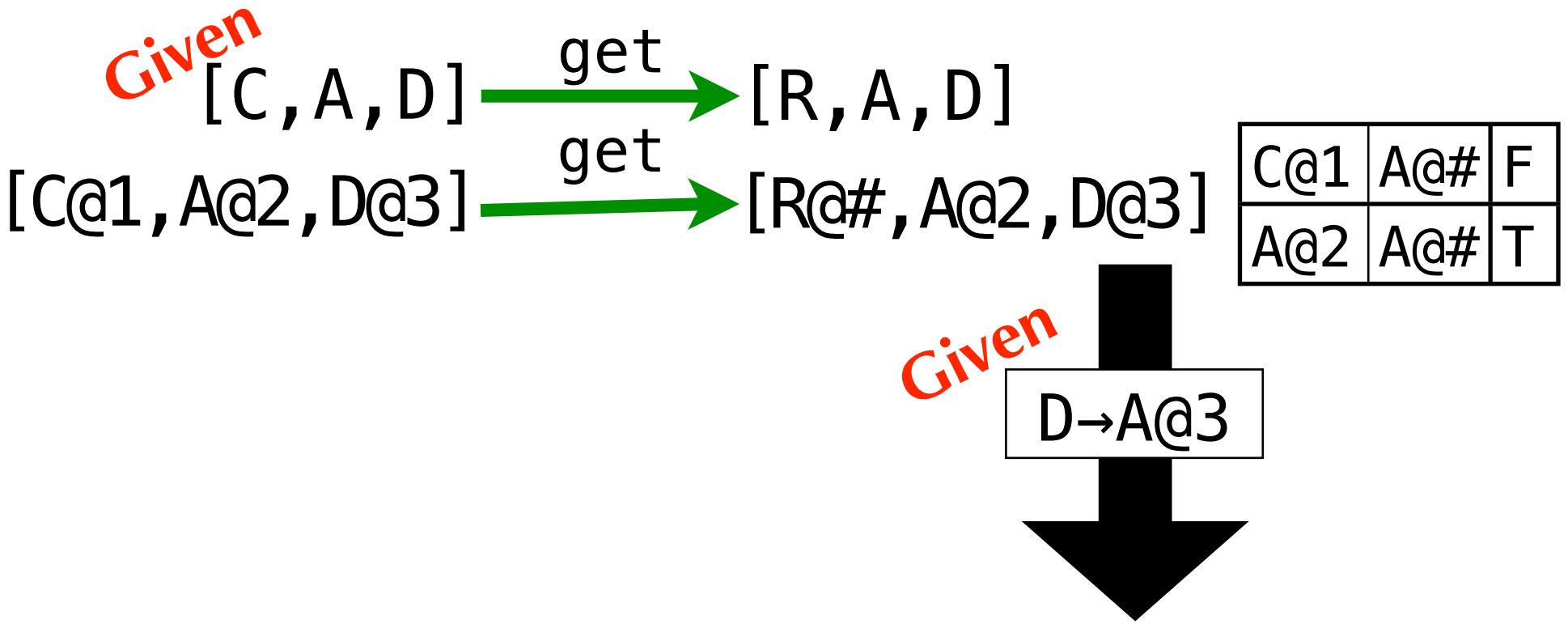
Given



C@1	A@#	F
A@2	A@#	T

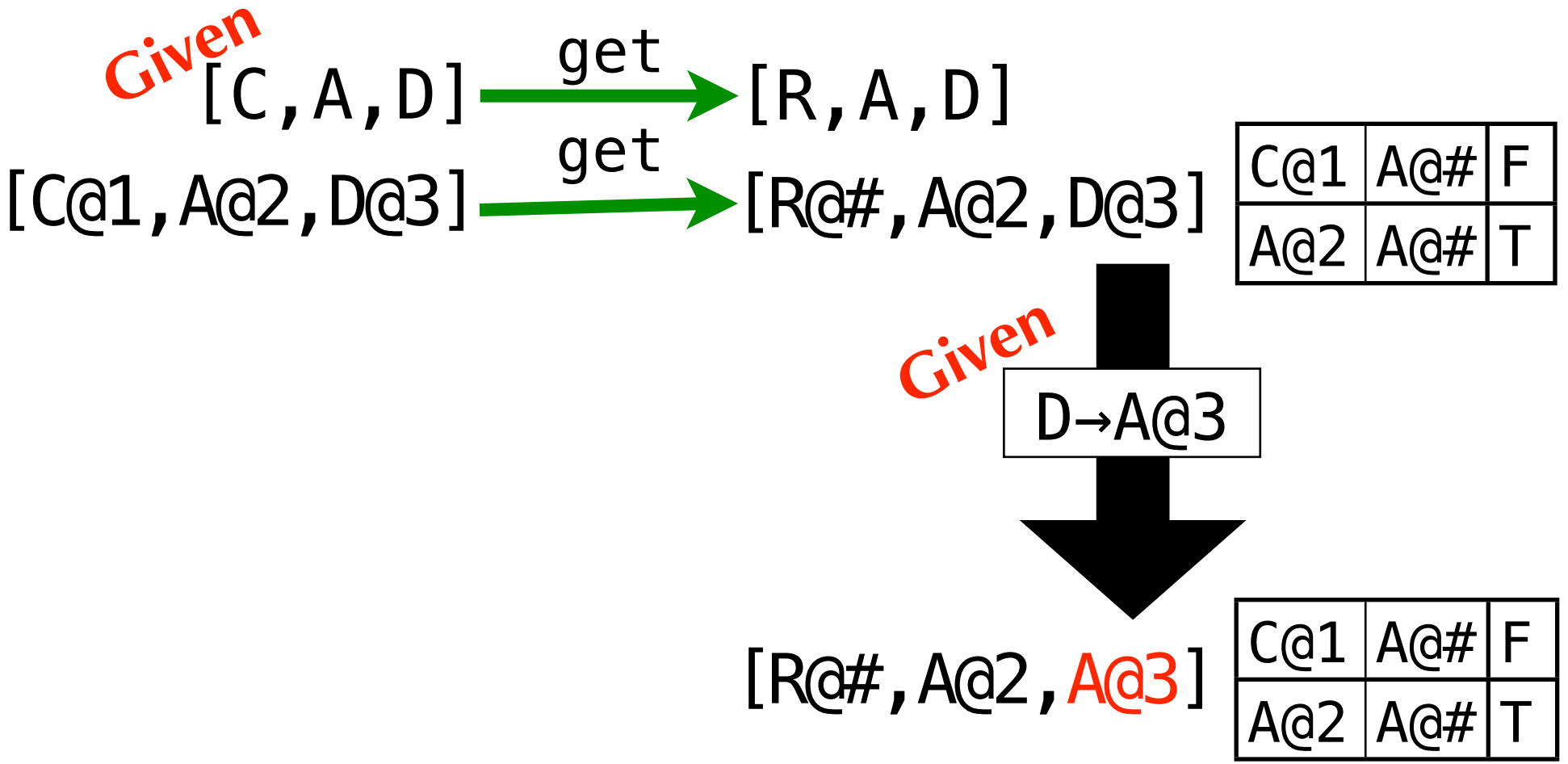
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



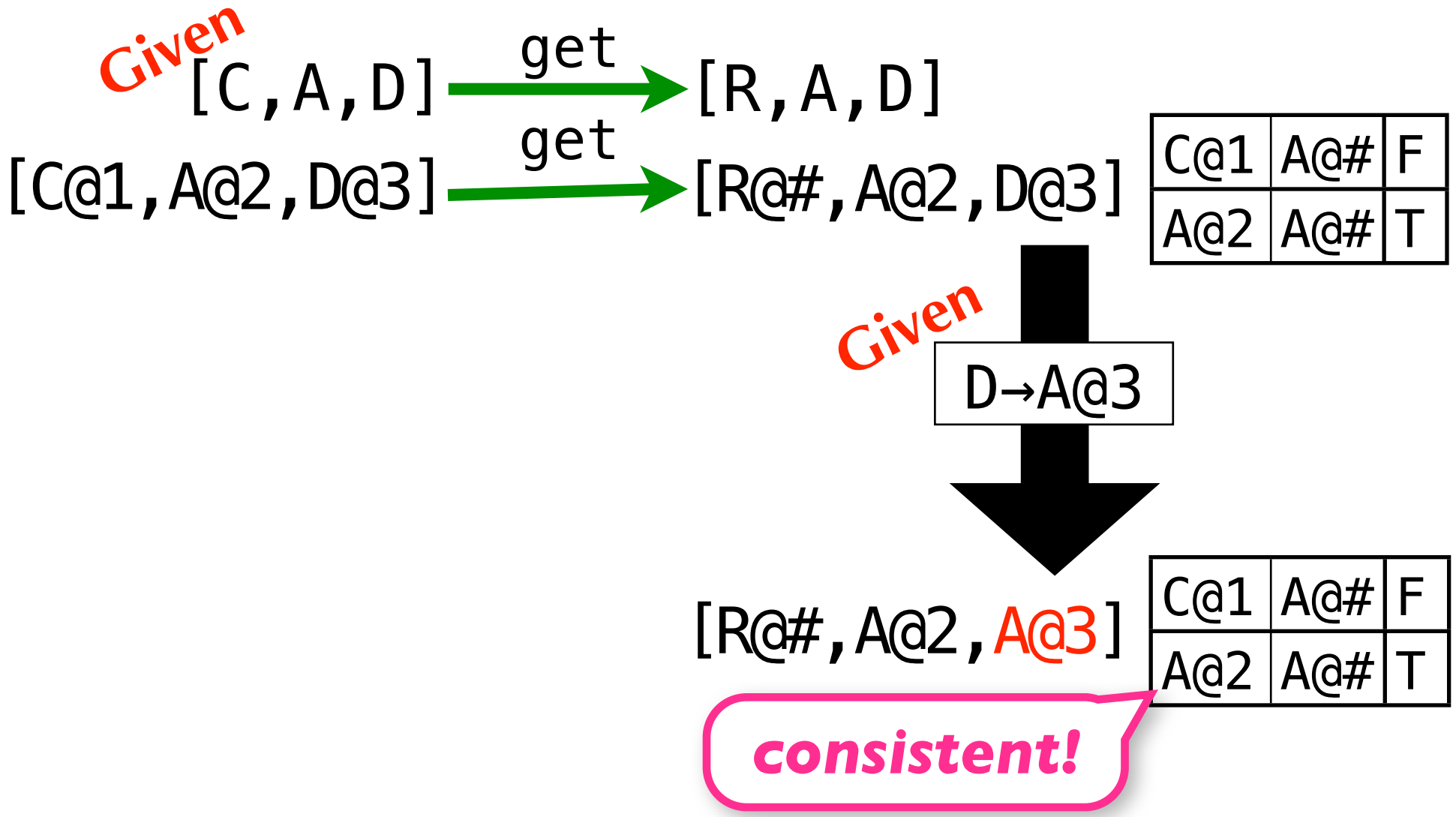
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



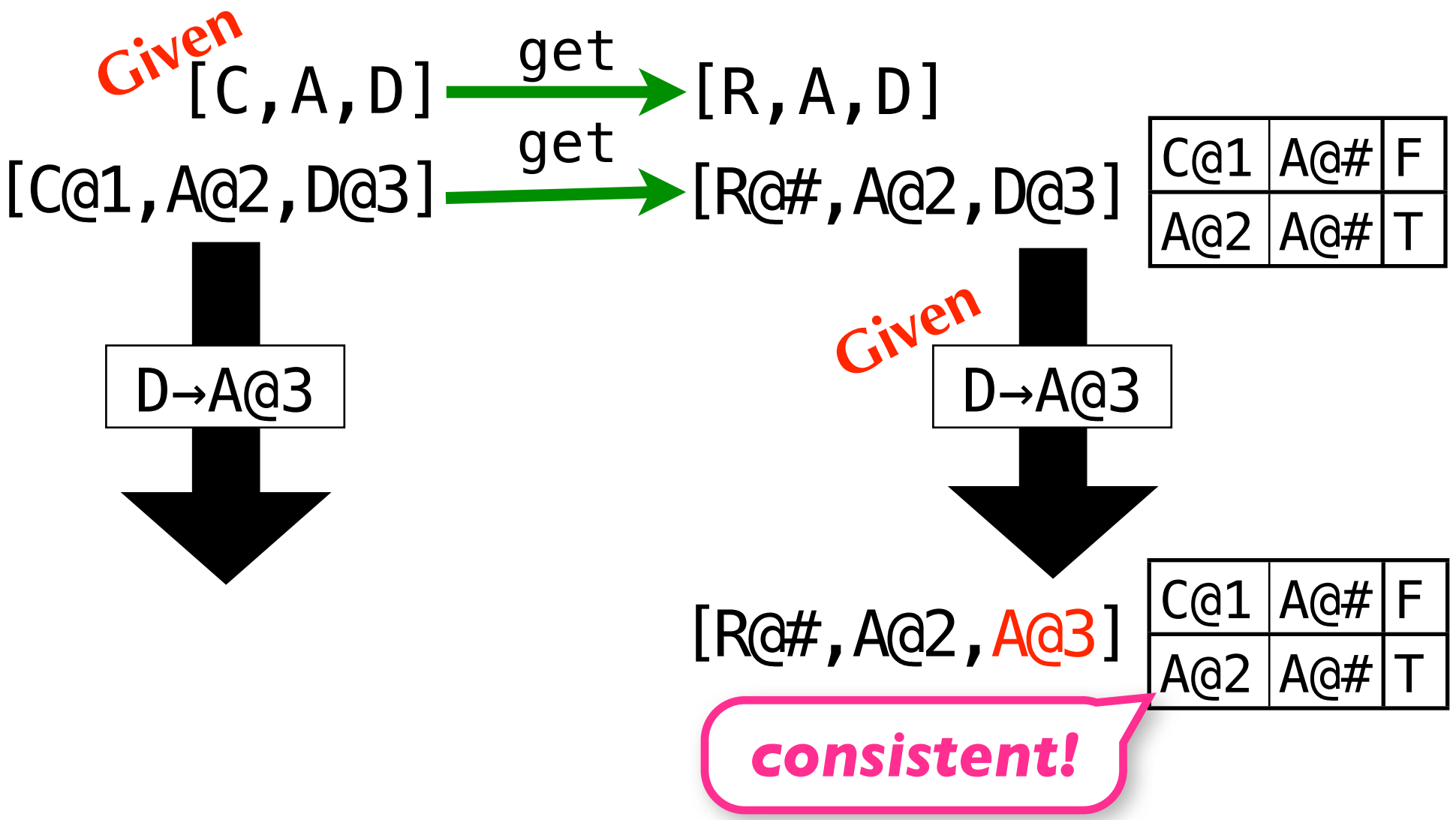
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



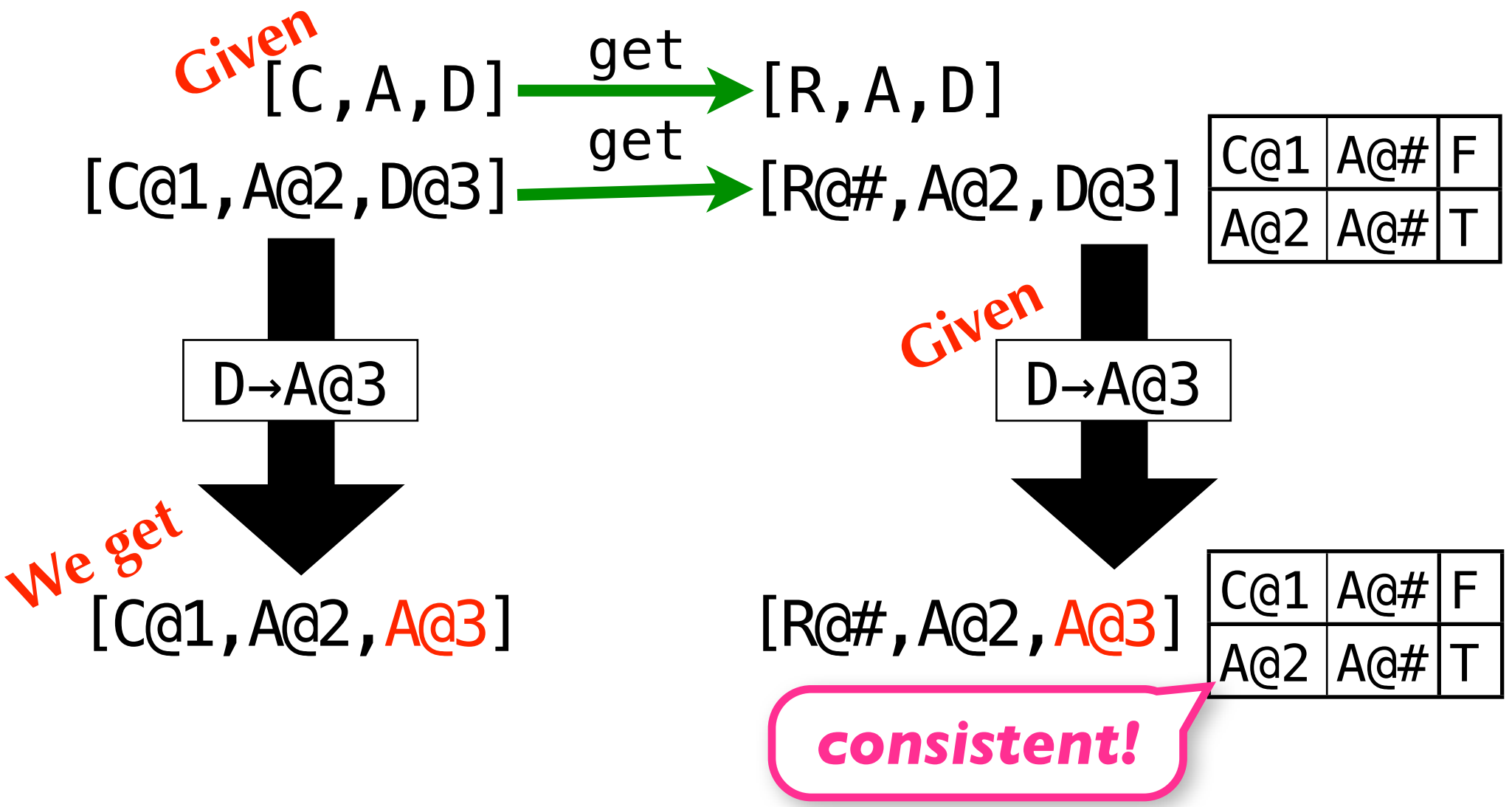
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



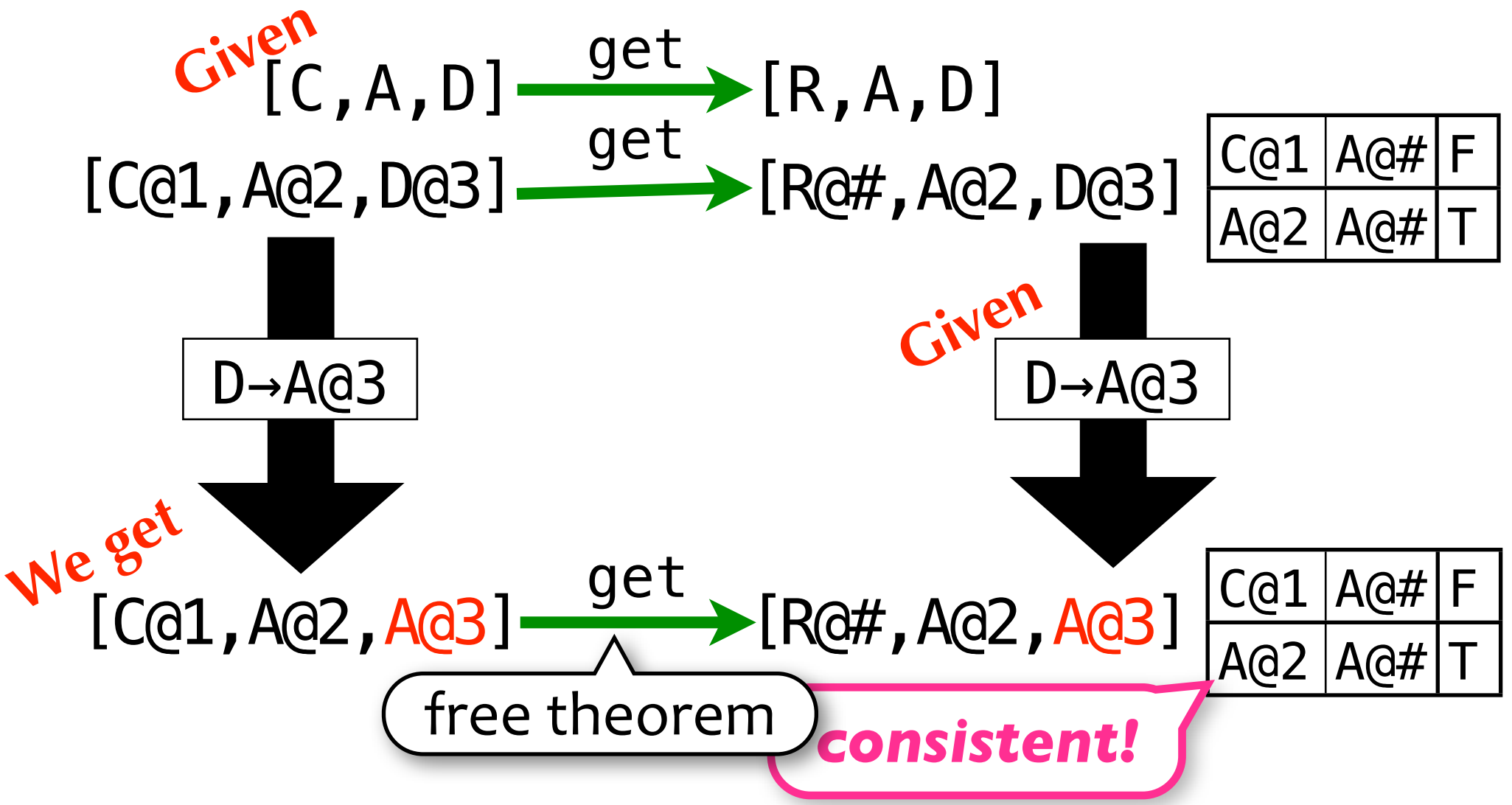
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



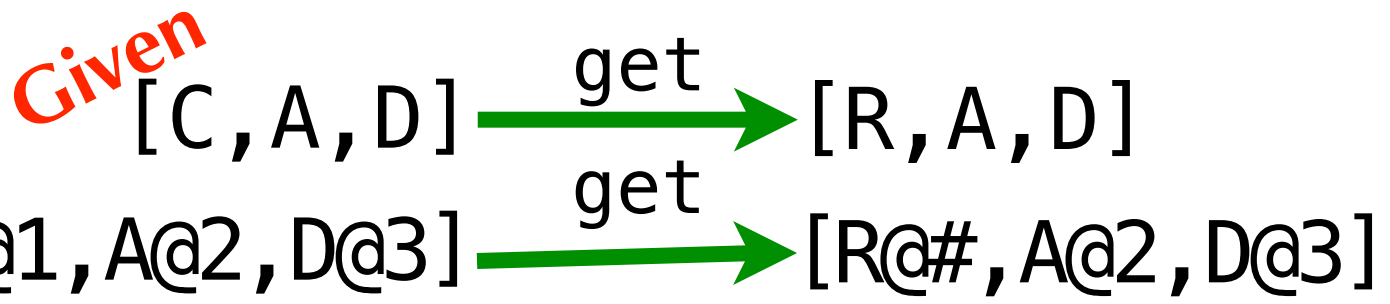
Our Bidirectionalization

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



Rejecting NG Updates

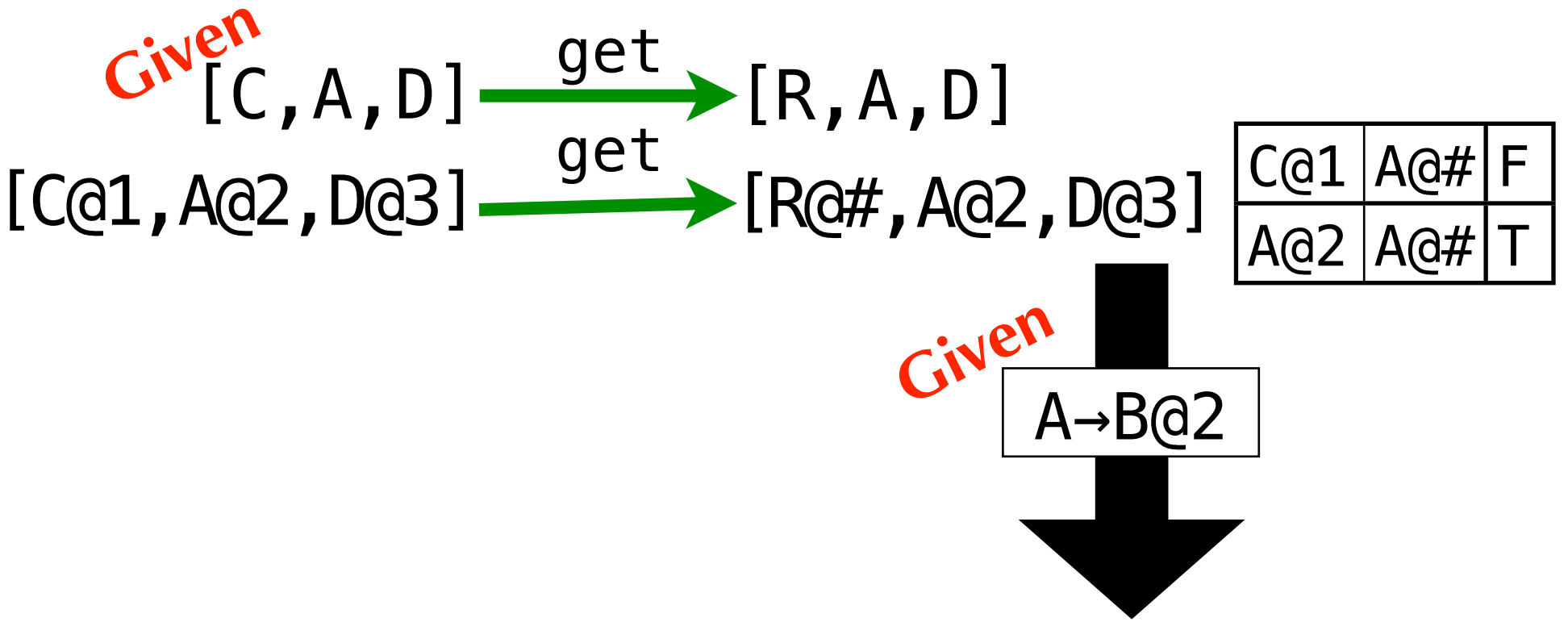
```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



C@1	A@#	F
A@2	A@#	T

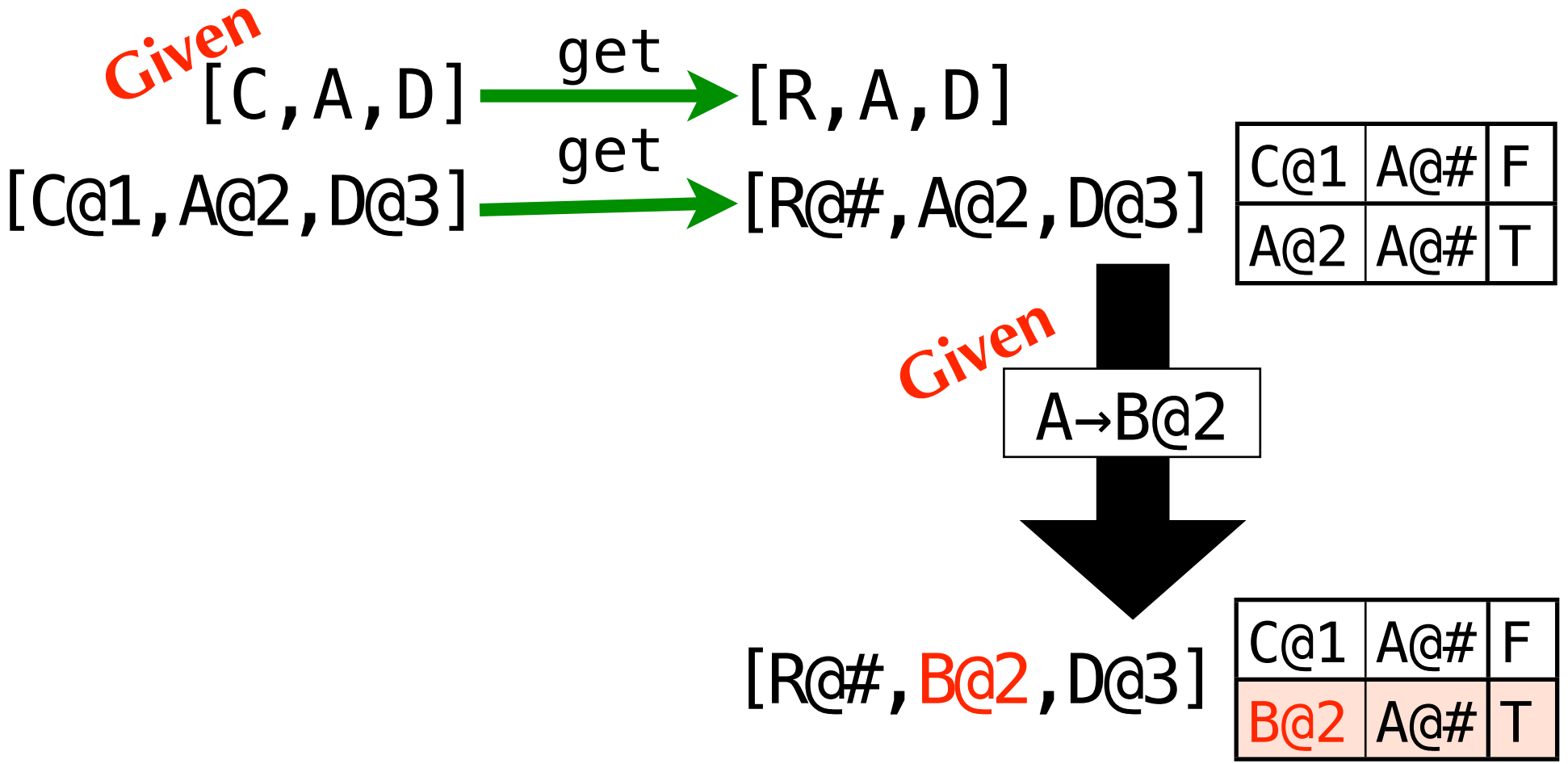
Rejecting NG Updates

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



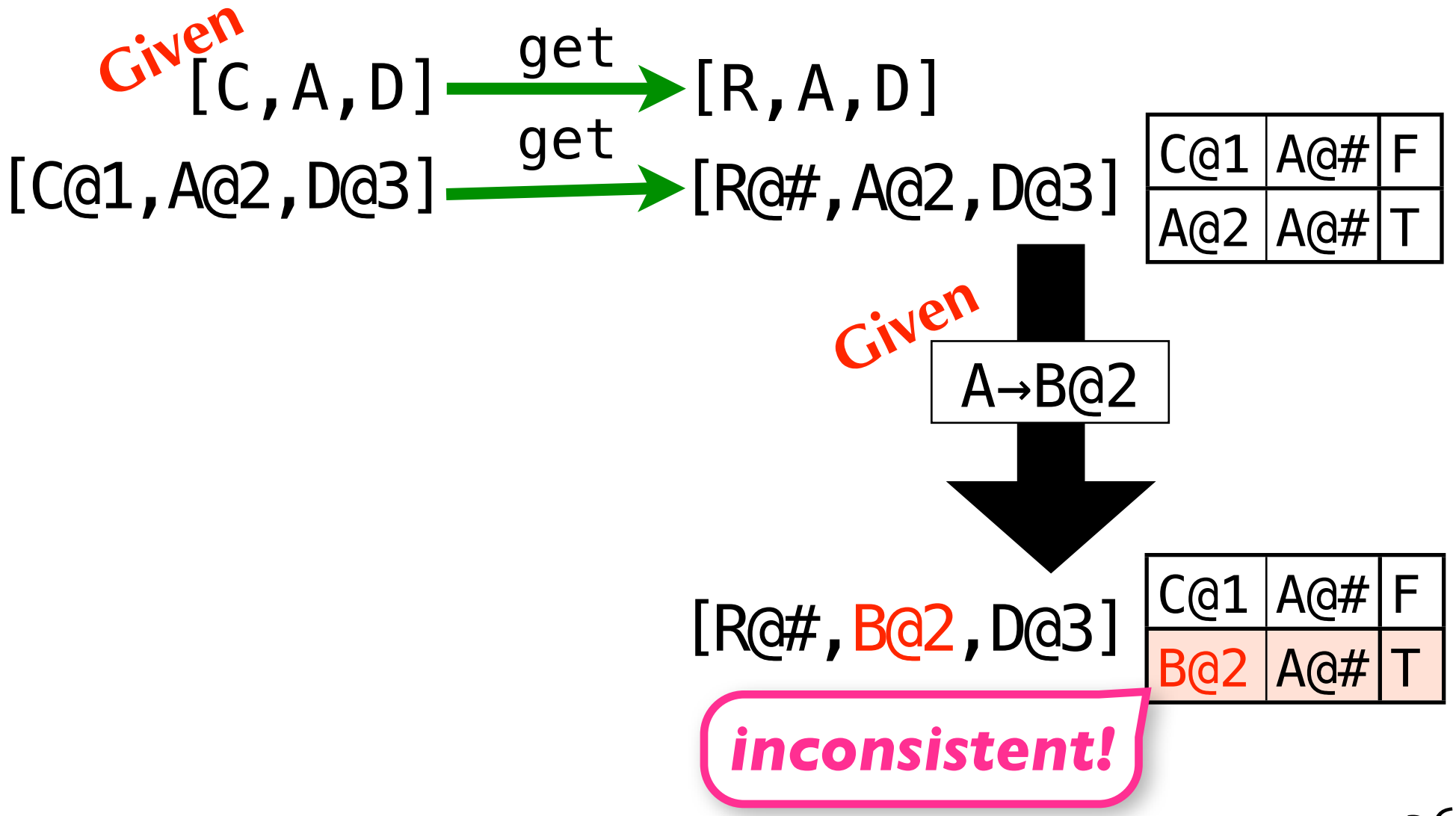
Rejecting NG Updates

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



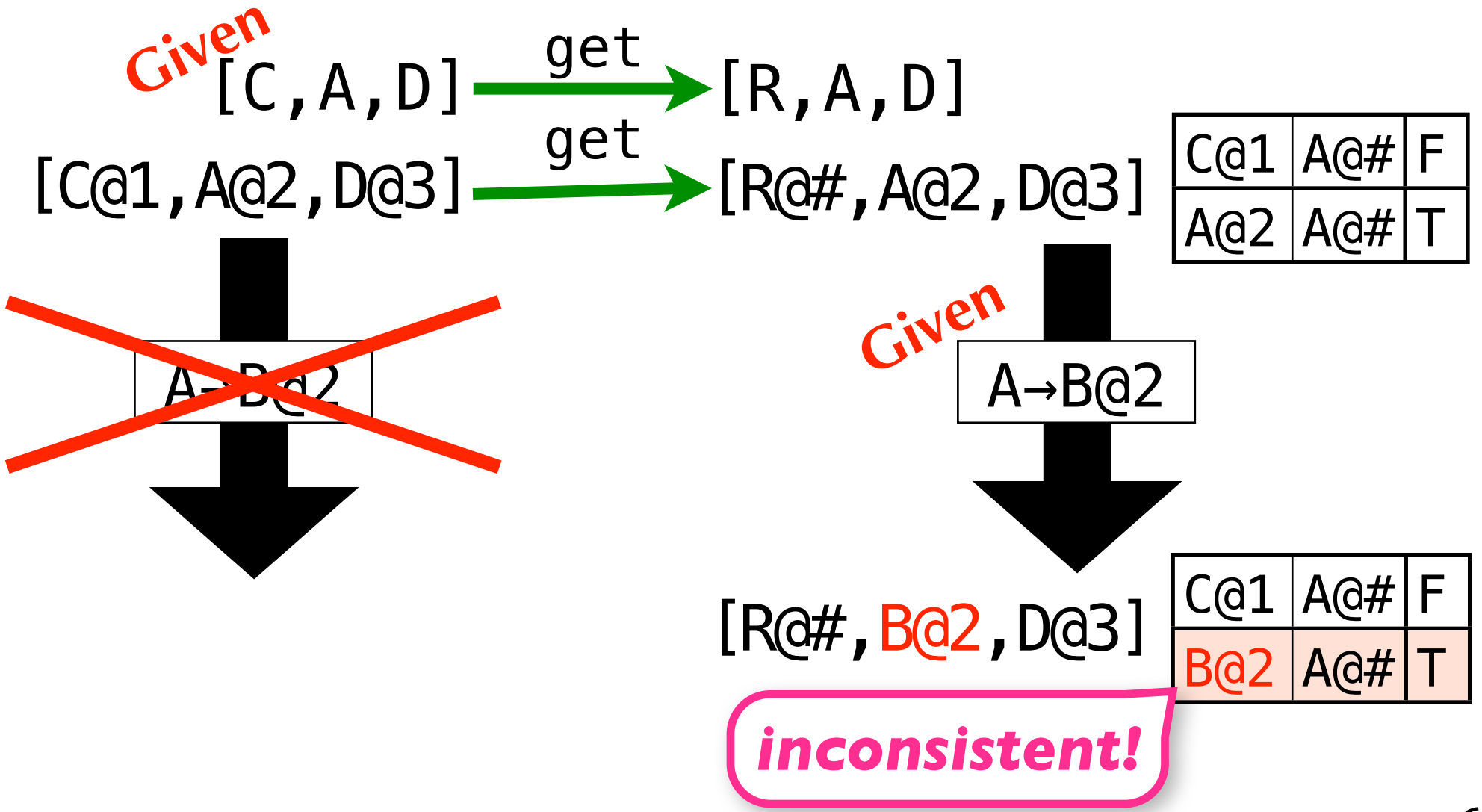
Rejecting NG Updates

```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



Rejecting NG Updates

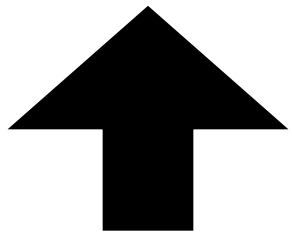
```
get :: ∀a.∀m.StringLike a m.[a] → m [a]
```



Generalization

```
get :: ∀a.∀m. PackM c a m => T1 a → m (T2 a)
```

```
class (Monad m, Pack c a) => PackM c a m where
   liftO :: Eq r => ([c] → r) → [a] → m r
class Pack c a where
   new :: c → a
```



generalize!

```
get :: ∀a.∀m. StringLike a m => [a] → m [a]
```

```
class Monad m => StringLike a m where
   new :: String → a
   eq  :: a → a → m Bool
```

Our Result

Result

We can derive a well-behaved bidirectional transformation from a get-function of type

$\forall a.\forall m.$ `PackM c a m =>` T_1 `a →` `m (`T_2 `a)`

```
class (Monad m, Pack c a) => PackM c a m where
    lift0 :: Eq r => ([c] → r) → [a] → m r
class Pack c a where
    new :: c → a
```

lifting comparison for ***runtime recording***

lifting creation of new elements

In the Paper ...

- Formal requirement on T_1 and T_2
- Formal proofs of the correctness

Result

We can derive a well-behaved bidirectional transformation from a get-function of type

```
∀a.∀m. PackM c a m => T₁ a → m (T₂ a)
```

- XML-transformation example
  - based on filters as in HaXML [Wallace&Runciman99]

Related Work

- Other approaches to construct bidirectional transformations
  - Combinator-based [Foster+07, Hu+04, ...]
  - Bidirectionalization [M+07, Voigtländer09,...]

Advantage in "bidirectionalization for free" (incl. ours)

user-friendliness

"get" can be ***arbitrary*** as long as it has a certain polymorphic type

Related Work

- Runtime-recording
  - Treatment of conditionals in [Foster+07, M+07]

Our Work

Reformalization of the idea in the context of "bidirectionalization for free" [Voigtländer09]

Conclusion

- An extension to "bidirectionalization for free" [Voigtländer 09]
  - Supporting various XML transformations
    - A ***type class*** to abstract concrete datatypes
    - ***Runtime-recording*** for well-behavedness

```
<bib>
 {
  for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

Conclusion

- An extension to "bidirectionalization for free" [Voigtländer 09]
  - Supporting various XML transformations
    - A ***type class*** to abstract concrete datatypes
    - ***Runtime-recording*** for well-behavedness

```
<bib>
 {
  for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
      { $b/title }
    </book>
 }
</bib>
```

implementation available!

https://bitbucket.org/kztk/cheap-b18n

Appendix

XML Transformation

- Assumption
  - XML documents are represented in the following rose trees

```
data Tree a = Node a [Tree a]
data L = E String | A String
       | T String
```

Idea

- Follow HaXml [Wallace&Runciman99]
  - Use filters
    - functions of type `Tree L -> [Tree L]`
  - In our situation, we use

```
PackM L a m =>
Tree a -> ListT m (Tree a)
```

XML Example

```
q1 :: PackM L a m => Tree a -> m (Tree a)
q1 t = pick
         $ do bs <- gather $ (keep /> (tag "book" >=> h) t
              return $ Node (new $ E "bib") bs
  where
    h b =
      do y <- (keep /> attr "year" /> keep) b
         t <- (keep /> tag "title") b
         p <- (keep /> tag "publisher" /> keep) b
         guardM $ lift $
           liftO2 gtInt (label y) (new $ T "1991")
         guardM $ lift $ liftO2
           liftO2 (==) (label p) (new $ T "Addison-Wesley")
         return $ Node (new $ E "book")
                       [Node (new $ A "year") [y]:t]
…
tag t = ofLabel (new $ E t)
…
gather :: Monad m => ListT m a -> ListT m [a]
pick :: Monad m => ListT m a -> m a
```