Isomorphic Data Type Transformations

Alessandro Coglio
Stephen Westfold

KESTREL INSTITUTE

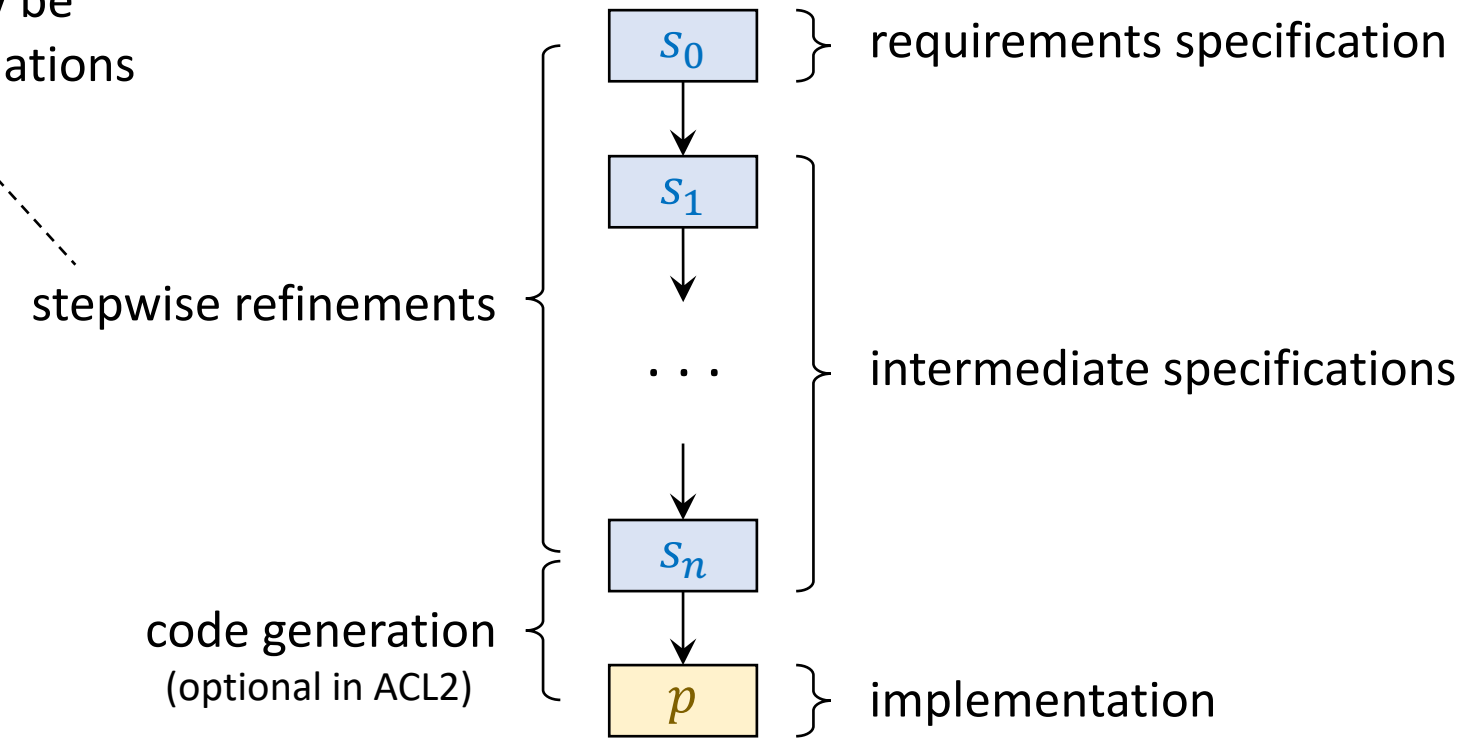
ACL2 WORKSHOP 2020

Isomorphic data type transformations are useful in program synthesis.

some of these may be isomorphic transformations

e.g.

- represent finite sets as repetition-free ordered lists
- turn unbounded integers into bounded integers (under preconditions)
- add redundant record components for caching
- change loop direction
- …

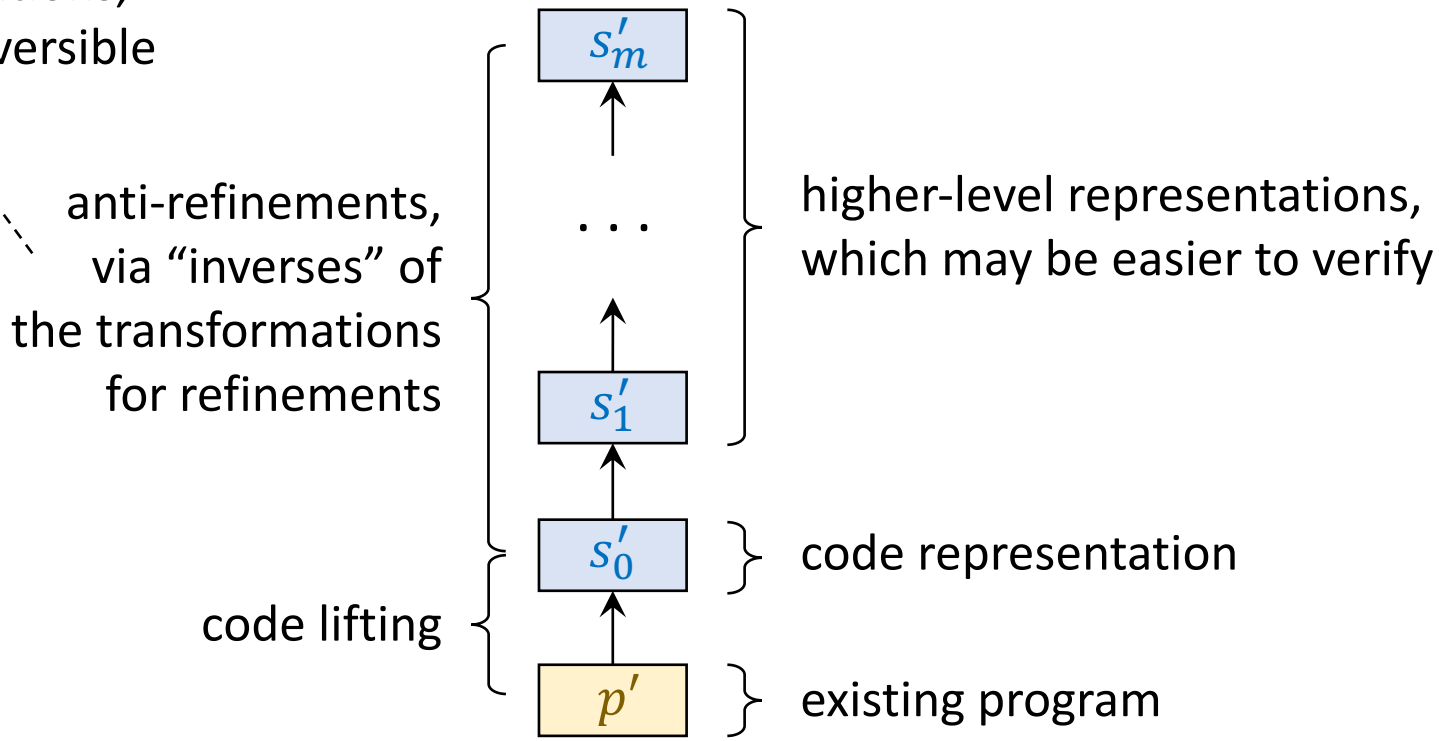


Isomorphic data type transformations are useful in program synthesis.
They are also useful in program analysis.

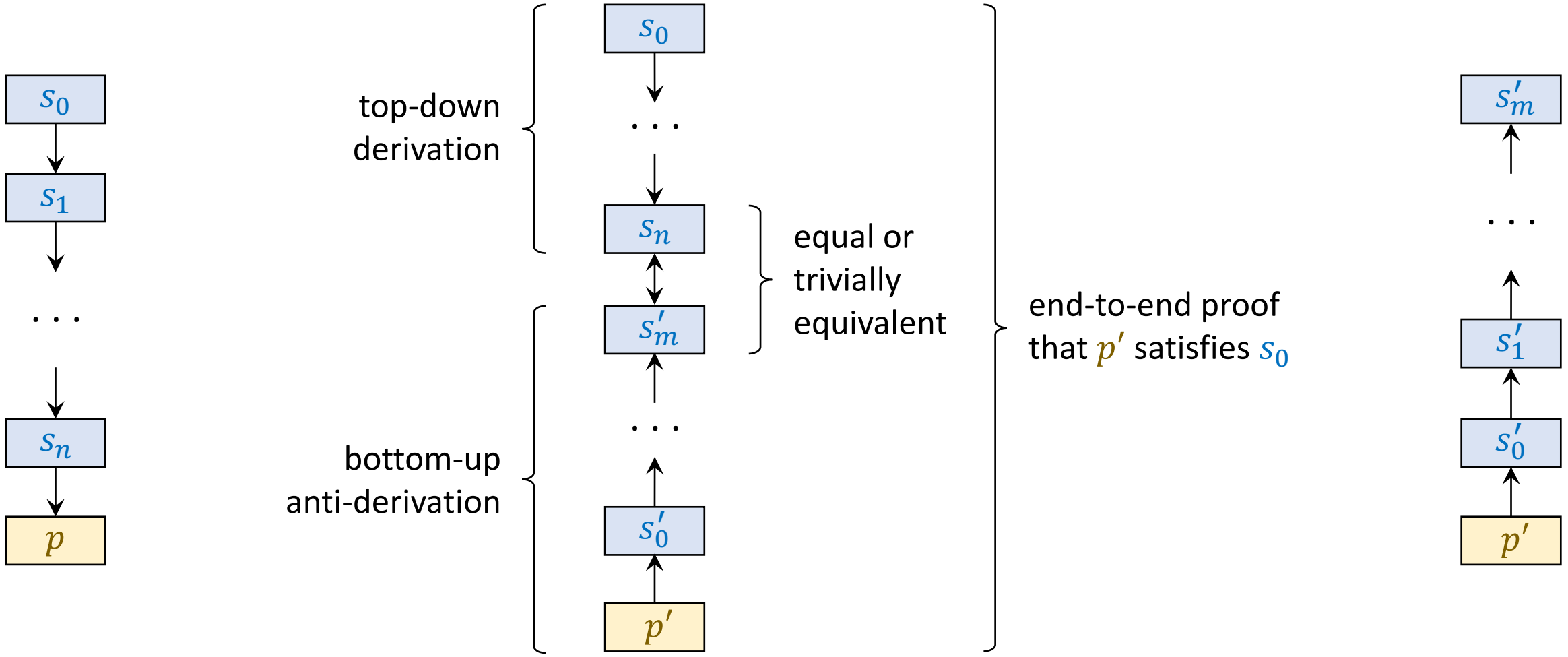
some of these may be
isomorphic transformations,
which are inherently reversible

e.g.

- represent repetition-free ordered lists as finite sets
- turn bounded integers into unbounded integers (under preconditions)
- remove redundant record components for caching
- change loop direction
- …

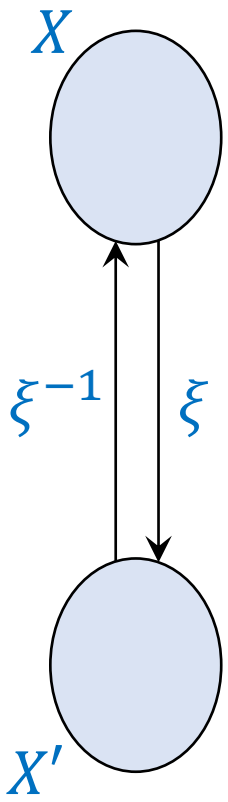


Isomorphic data type transformations are useful in program synthesis.
They are also useful in program analysis, as well as in analysis-by-synthesis.



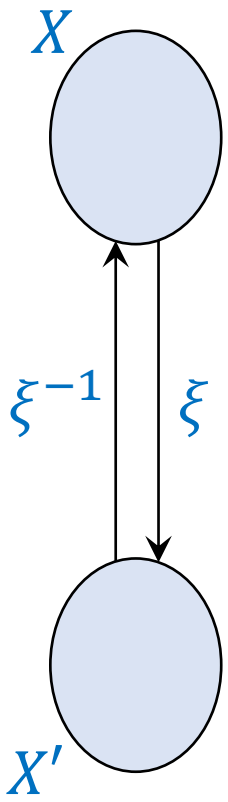
Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.

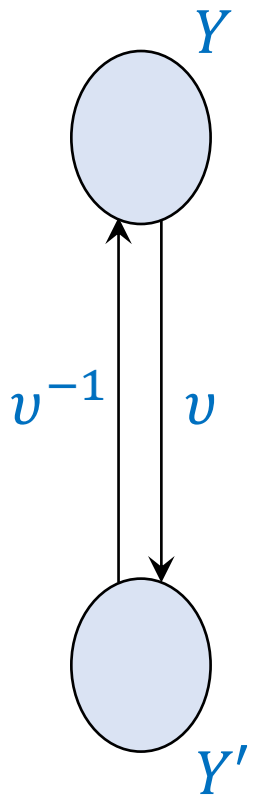
$$\xi^{-1} \circ \xi = id_X$$
$$\xi \circ \xi^{-1} = id_{X'}$$



Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.
Consider two isomorphic sets (data types) Y and Y' with $\upsilon : Y \longrightarrow Y'$ and $\upsilon^{-1} : Y' \longrightarrow Y$.

$$\xi^{-1} \circ \xi = id_X$$
$$\xi \circ \xi^{-1} = id_{X'}$$

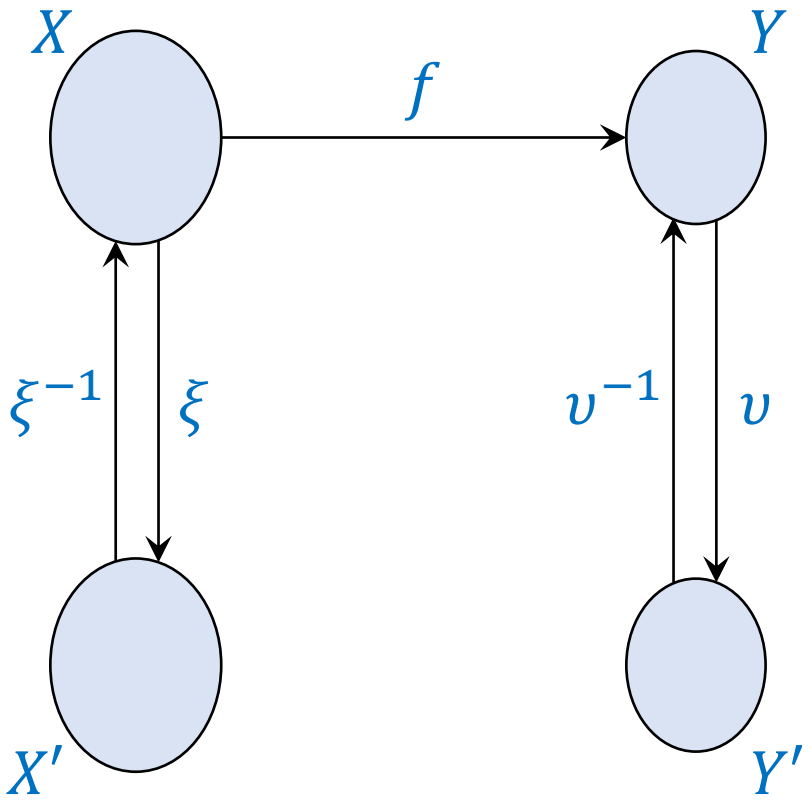




$$\upsilon^{-1} \circ \upsilon = id_Y$$
$$\upsilon \circ \upsilon^{-1} = id_{Y'}$$

Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.
Consider two isomorphic sets (data types) Y and Y' with $\upsilon : Y \longrightarrow Y'$ and $\upsilon^{-1} : Y' \longrightarrow Y$.
Consider a function $f : X \longrightarrow Y$, a computation from inputs of type X to outputs of type Y.

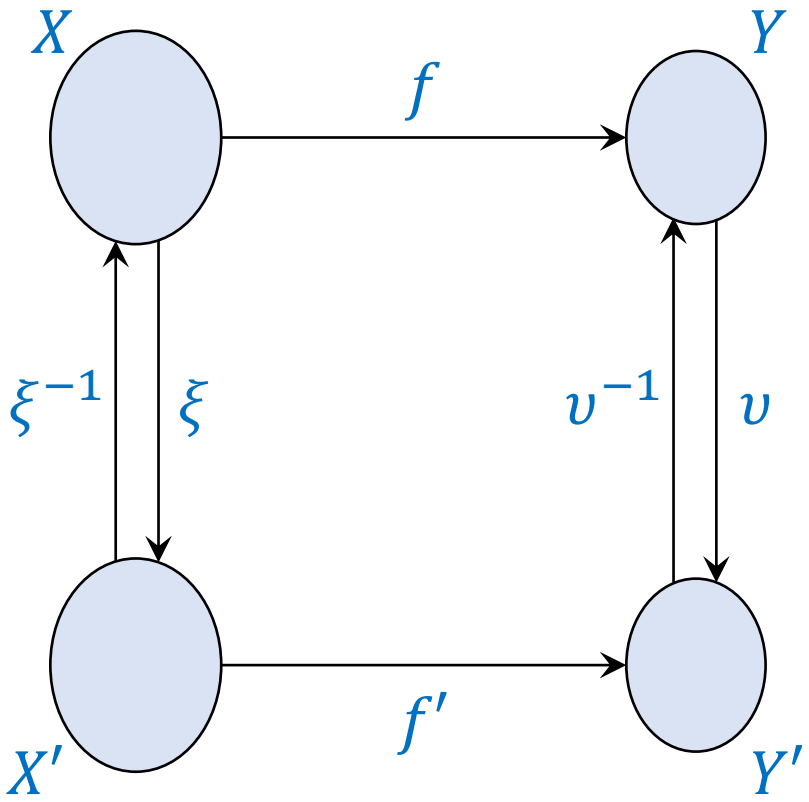
$$\xi^{-1} \circ \xi = id_X$$
$$\xi \circ \xi^{-1} = id_{X'}$$



$$\upsilon^{-1} \circ \upsilon = id_Y$$
$$\upsilon \circ \upsilon^{-1} = id_{Y'}$$

Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.
Consider two isomorphic sets (data types) Y and Y' with $\upsilon : Y \longrightarrow Y'$ and $\upsilon^{-1} : Y' \longrightarrow Y$.
Consider a function $f : X \longrightarrow Y$, a computation from inputs of type X to outputs of type Y.
We can mechanically construct a function $f' : X' \longrightarrow Y'$ that makes the diagram commute.

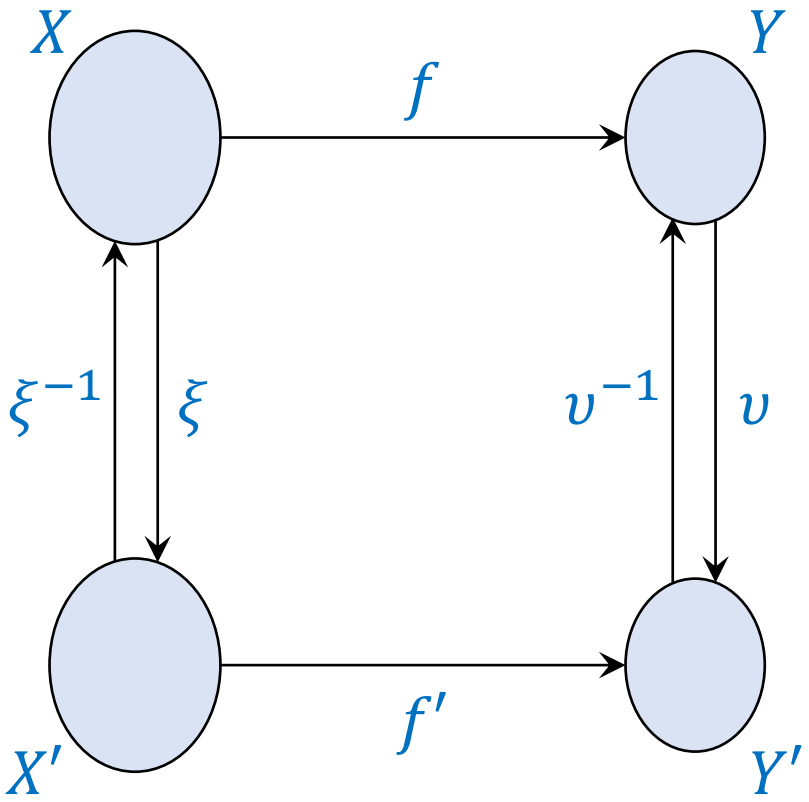
$$\xi^{-1} \circ \xi = id_X$$
$$\xi \circ \xi^{-1} = id_{X'}$$



$$\upsilon^{-1} \circ \upsilon = id_Y$$
$$\upsilon \circ \upsilon^{-1} = id_{Y'}$$

$$f' = \upsilon \circ f \circ \xi^{-1} \quad \Leftrightarrow \quad f = \upsilon^{-1} \circ f' \circ \xi$$

Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.
Consider two isomorphic sets (data types) Y and Y' with $\upsilon : Y \longrightarrow Y'$ and $\upsilon^{-1} : Y' \longrightarrow Y$.
Consider a function $f : X \longrightarrow Y$, a computation from inputs of type X to outputs of type Y.
We can mechanically construct a function $f' : X' \longrightarrow Y'$ that makes the diagram commute.

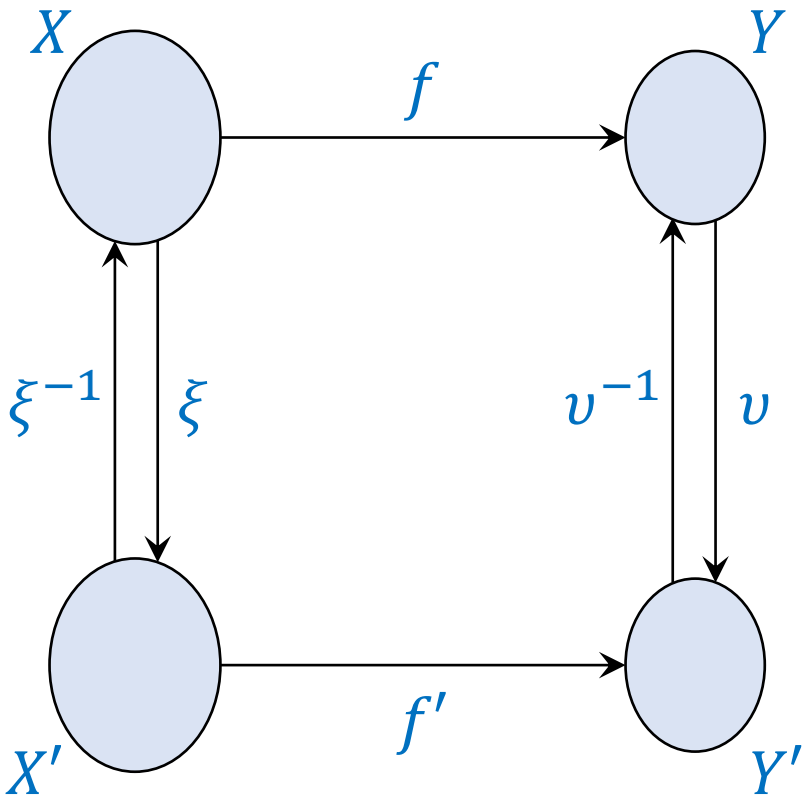


$$f' = \upsilon \circ f \circ \xi^{-1}$$

we could just define f' like this,
but that is not very interesting

$$f' \equiv \upsilon \circ f \circ \xi^{-1}$$

Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.
Consider two isomorphic sets (data types) Y and Y' with $\upsilon : Y \longrightarrow Y'$ and $\upsilon^{-1} : Y' \longrightarrow Y$.
Consider a function $f : X \longrightarrow Y$, a computation from inputs of type X to outputs of type Y.
We can mechanically construct a function $f' : X' \longrightarrow Y'$ that makes the diagram commute.



$$f' = \upsilon \circ f \circ \xi^{-1}$$

$$f(x) \equiv \textbf{if } a(x) \textbf{ then } b(x) \textbf{ else } c(x, f(d(x)))$$

representative recursive definition

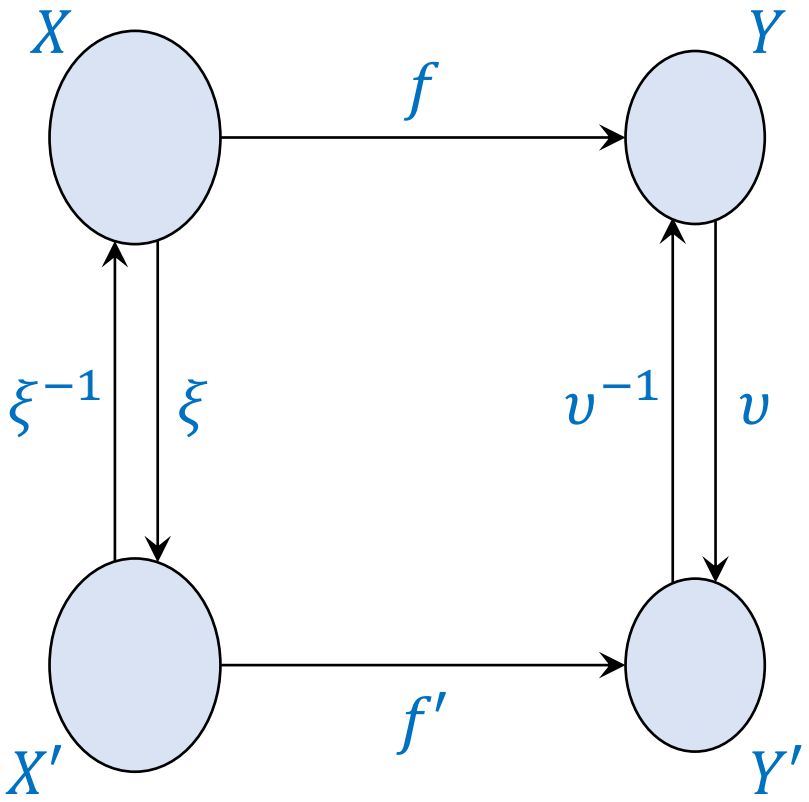
$a \subseteq X \qquad b : X \longrightarrow Y \qquad c : X \times Y \longrightarrow Y \qquad d : X \longrightarrow X$

$\vdash \neg a(x) \Longrightarrow \mu(d(x)) \prec \mu(x) \qquad f$ terminates

keep the same structure
and add the conversions

$$f'(x') \equiv \textbf{if } a(\xi^{-1}(x')) \textbf{ then } \upsilon(b(\xi^{-1}(x'))) \textbf{ else } \upsilon(c(\xi^{-1}(x'), \upsilon^{-1}(f'(\xi(d(\xi^{-1}(x')))))))$$

Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.
Consider two isomorphic sets (data types) Y and Y' with $\upsilon : Y \longrightarrow Y'$ and $\upsilon^{-1} : Y' \longrightarrow Y$.
Consider a function $f : X \longrightarrow Y$, a computation from inputs of type X to outputs of type Y.
We can mechanically construct a function $f' : X' \longrightarrow Y'$ that makes the diagram commute.



$$f(x) \equiv \textbf{if } a(x) \textbf{ then } b(x) \textbf{ else } c(x, f(d(x)))$$ } representative recursive definition

$a \subseteq X \qquad b : X \longrightarrow Y \qquad c : X \times Y \longrightarrow Y \qquad d : X \longrightarrow X$

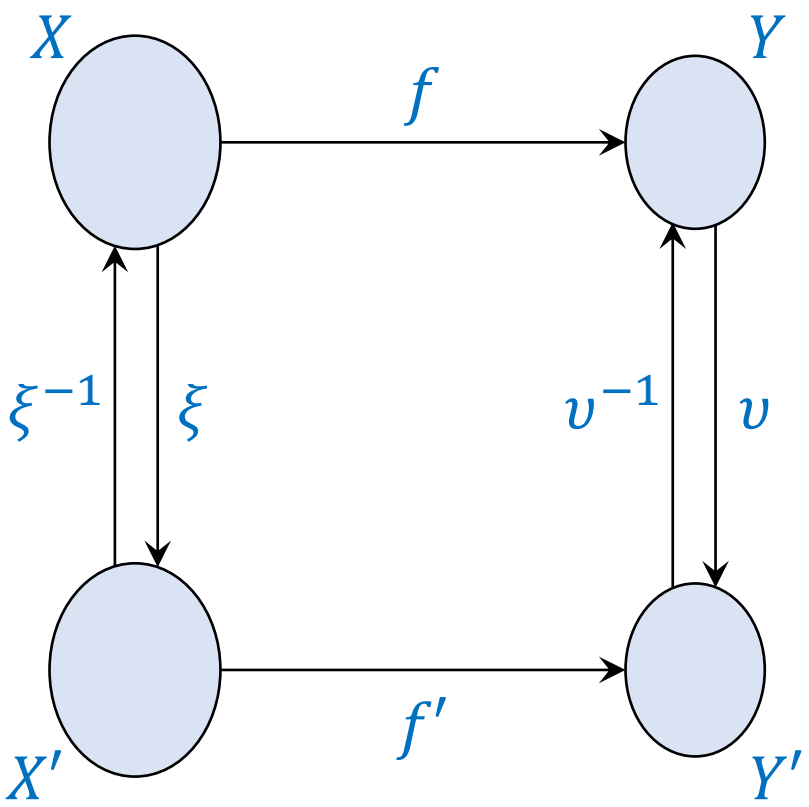
$\vdash \neg a(x) \Longrightarrow \mu(d(x)) \prec \mu(x)$ $\qquad$ f terminates

keep the same structure
and add the conversions } automatic

$$f'(x') \equiv \textbf{if } a(\xi^{-1}(x')) \textbf{ then } \upsilon(b(\xi^{-1}(x'))) \textbf{ else } \upsilon(c(\xi^{-1}(x'), \upsilon^{-1}(f'(\xi(d(\xi^{-1}(x')))))))$$

$\mu' \equiv \mu \circ \xi^{-1}$ $\qquad$ f' terminates because f does

$\vdash f' = \upsilon \circ f \circ \xi^{-1}$ <---- by induction



$$\vdash f' = \upsilon \circ f \circ \xi^{-1}$$

$$f(x) \equiv \textbf{if } a(x) \textbf{ then } b(x) \textbf{ else } c(x, f(d(x)))$$

keep the same structure and add the conversions } automatic

$$f'(x') \equiv \textbf{if } a(\xi^{-1}(x')) \textbf{ then } \upsilon(b(\xi^{-1}(x'))) \textbf{ else } \upsilon(c(\xi^{-1}(x'), \upsilon^{-1}(f'(\xi(d(\xi^{-1}(x')))))))$$

expand the definitions and rewrite/simplify } user-guided

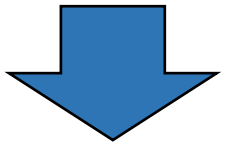
$$f''(x') \equiv \textbf{if } a'(x') \textbf{ then } b'(x') \textbf{ else } c'(x', f''(d'(x')))$$

goal: no trace of $X, Y, \xi, \xi^{-1}, \upsilon, \upsilon^{-1}$

$$\vdash f'' = f'$$

This is a general method:
automatically create an isomorphic version
and semi-automatically rewrite/simplify it.
We can do it for f, f_1, f_2, etc.,
obtaining f', f'', f_1', f_1'', f_2', f_2'', etc.,

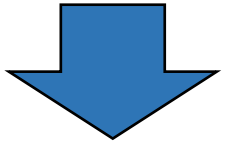
$f(x) \equiv \ \ldots$



keep the same structure
and add the conversions

$f'(x') \equiv \ \ldots$

$\vdash f = v^{-1} \circ f' \circ \xi$



expand the definitions
and rewrite/simplify

$f''(x') \equiv \ \ldots$

Consider a function g that calls f, f_1, f_2, etc.
We can apply the same general method to g.
If g manipulates the data being transformed
only through f, f_1, f_2, etc., we can automate
the rewriting/simplification step as well.

$g(\ldots) \equiv \ \ldots \ f(\ldots) \ \ldots$

keep the same structure
and add the conversions

$g'(\ldots) \equiv \ \ldots \ v(f(\xi^{-1}(\ldots))) \ \ldots$

expand the definitions
and rewrite/simplify

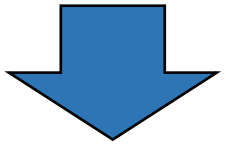
$g''(\ldots) \equiv \ \ldots$

This is a general method:
automatically create an isomorphic version
and semi-automatically rewrite/simplify it.
We can do it for f, f_1, f_2, etc.,
obtaining f', f'', f_1', f_1'', f_2', f_2'', etc.,

$f(x) \equiv \ldots$

keep the same structure
and add the conversions

$f'(x') \equiv \ldots$

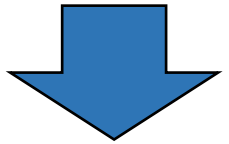


expand the definitions
and rewrite/simplify

$f''(x') \equiv \ldots$

Consider a function g that calls f, f_1, f_2, etc.
We can apply the same general method to g.
If g manipulates the data being transformed
only through f, f_1, f_2, etc., we can automate
the rewriting/simplification step as well.
And we can do everything in one step.

$g(\ldots) \equiv \ldots f(\ldots) \ldots$



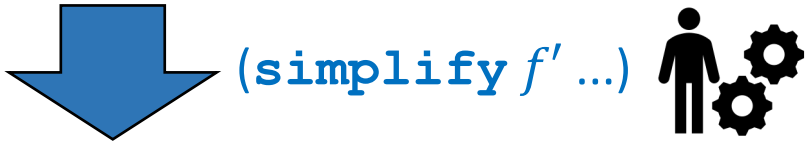
keep the same structure
and replace f with f' etc.

$g'(\ldots) \equiv \ldots f'(\ldots) \ldots$

We use **`isodata`** to initiate the isomorphic transformation.

$f(x) \equiv \ldots$

$f'(x') \equiv \ldots$



$f''(x') \equiv \ldots$

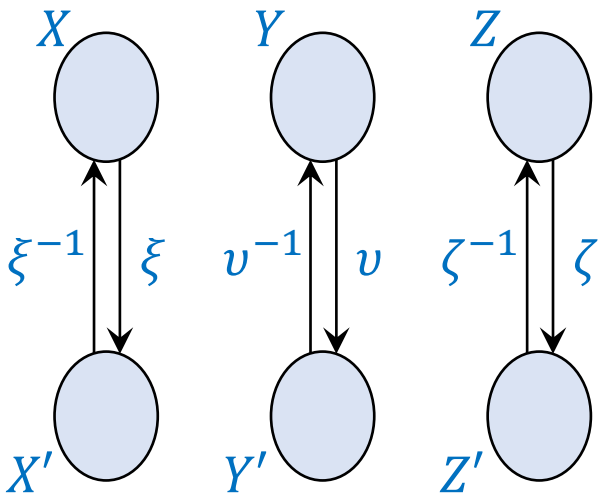
The **`simplify`** transformation was described at ACL2-2017.

We use **`propagate-iso`** to propagate the isomorphic transformation.

$g(\ldots) \equiv \ldots f(\ldots) \ldots$

$g'(\ldots) \equiv \ldots f'(\ldots) \ldots$

We use **defiso** to establish the isomorphic mappings.



(**defiso** $X\ X'\ \xi\ \xi^{-1}$ ...)

(**defiso** $Y\ Y'\ \upsilon\ \upsilon^{-1}$ ...)

(**defiso** $Z\ Z'\ \zeta\ \zeta^{-1}$ ...)

and for other types

We use **isodata** to initiate the isomorphic transformation.

$f(x) \equiv \ \ldots$

(**isodata** f ...)

$f'(x') \equiv \ \ldots$

(**simplify** f' ...)

$f''(x') \equiv \ \ldots$

and for f_1, f_2, etc.

We use **propagate-iso** to propagate the isomorphic transformation.

$g(\ldots) \equiv \ \ldots\ f(\ldots)\ \ldots$

(**propagate-iso** g ...)

$g'(\ldots) \equiv \ \ldots\ f'(\ldots)\ \ldots$

and for g_1, g_2, etc.

`propagate-iso`

1. Find events to propagate to
   - User supplied limits
   - Dependent events: function definitions and theorems
2. Type analysis: which arguments and results to be transformed
   - Arguments: guards
   - Results: typing theorems and body of definition
3. Dependent isomorphisms
   - Subtypes, record/product types, recursive types: e.g. list, map types
4. Translation: substitution
   - Add isomorphism theorems for newly generated functions
5. Hints. Hard to guarantee they will work
   - Three rulesets: forward, backward, general (typing, defiso rules)
   - Allow user to augment or override automatically generated hints

Dependent Isomorphism: Deriving isomorphism from predicate

Consider two isomorphic sets (data types) X and X' with $\xi : X \longrightarrow X'$ and $\xi^{-1} : X' \longrightarrow X$.

Consider a predicate $AllX(l) \equiv \textbf{if } atom(l) \textbf{ then } null(l) \textbf{ else } X(car(l)) \wedge AllX(cdr(l))$.

Then define predicate $AllX'(l) \equiv \textbf{if } atom(l) \textbf{ then } null(l) \textbf{ else } X'(car(l)) \wedge AllX'(cdr(l))$.

We want to find definitions for the isomorphisms $All\xi : AllX \longrightarrow AllX'$ and $All\xi^{-1} : AllX' \longrightarrow AllX$.

$null(l) \qquad X'(car(l)) \wedge AllX'(cdr(l))$

$All\xi(l) \equiv \textbf{if } atom(l) \textbf{ then } b(l) \textbf{ else } c(X(car(l)), AllX(cdr(l)))$

$null(b(l)) \Longrightarrow b(l) = nil$

$\textbf{if } atom(l) \textbf{ then } nil \textbf{ else } E$

$E = c(X(car(l)), AllX(cdr(l)))$

$\wedge\, consp(E) \wedge X'(car(E)) \wedge AllX'(cdr(E))$

$\therefore E = cons(\xi(car(l)), All\xi(cdr(l)))$

$All\xi(l) \equiv \textbf{if } atom(l) \textbf{ then } nil \textbf{ else } cons(\xi(car(l)), All\xi(cdr(l)))$

Consider two isomorphic sets (data types) `P` and `P'` with `P-to-P': P ⟶ P'`

```
(defun P-map-p (m)
  (if (atom m)
      (null m)
    (and (consp (car m))
         (P (caar m))
         (natp (cdar m))
         (P-map-p (cdr m)))))
```

```
(defun P'-map-p (m)
  (if (atom m)
      (null m)
    (and (consp (car m))
         (P' (caar m))
         (natp (cdar m))
         (P'-map-p (cdr m)))))
```

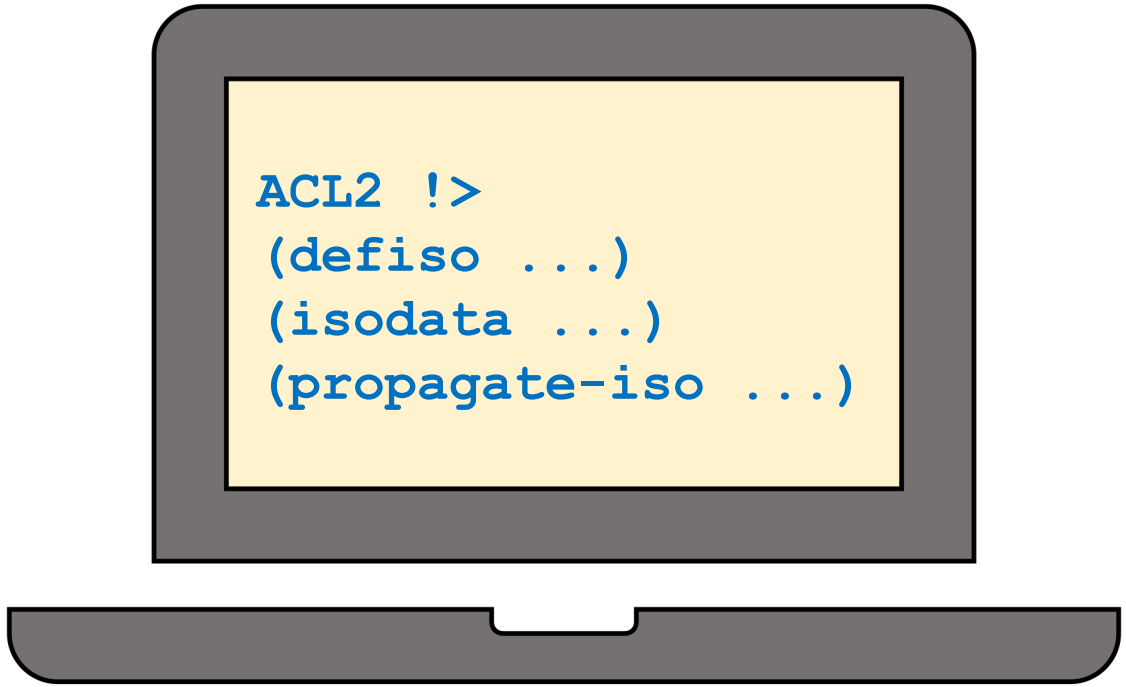
Derive `P-map-to-P'-map` else clause

```
(consp (car m))    --> (cons (cons ? ?) ?)
(P' (caar m))      --> (cons (cons (P-to-P' (caar m)) ?) ?)
(natp (cdar m))    --> (cons (cons ? (cdar m)) ?)
(P'-map-p (cdr m))--> (cons (cons ? ?) (P-map-to-P'-map (cdr m)))
Combined:             (cons (cons (P-to-P' (caar m)) (cdar m)) (P-map-to--map (cdr m)))
```

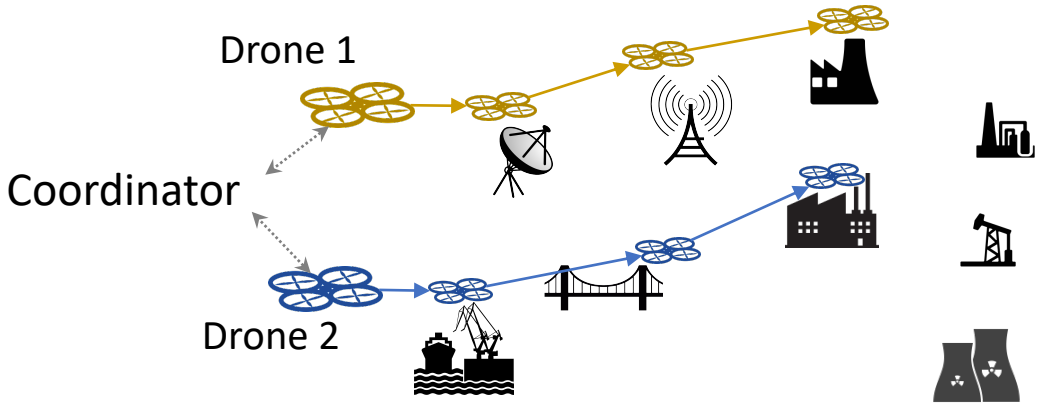
identity isomorphism

```
(defun P-map-to-P'-map (m)
  (if (atom m)
      nil
    (cons (cons (P-to-P' (caar m))
                (cdar m))
          (P-map-to-P'-map (cdr m)))))
```

Demo: efficient value caching with invariant maintenance.



Drone planner



- Set of drones has to visit a set of sites
- Partial plan then execute cycle until all sites visited
- Each drone produces candidate plans for itself
- Coordinator filters plans to minimize redundancy
- Each drone has a state
- System state is a list of drone states plus coordinator state