Synthesizing Transformations on Hierarchically Structured Data

Navid Yaghmazadeh, *Christian Klinger, Isil Dillig, Swarat Chaudhuri*

THE UNIVERSITY OF TEXAS AT AUSTIN

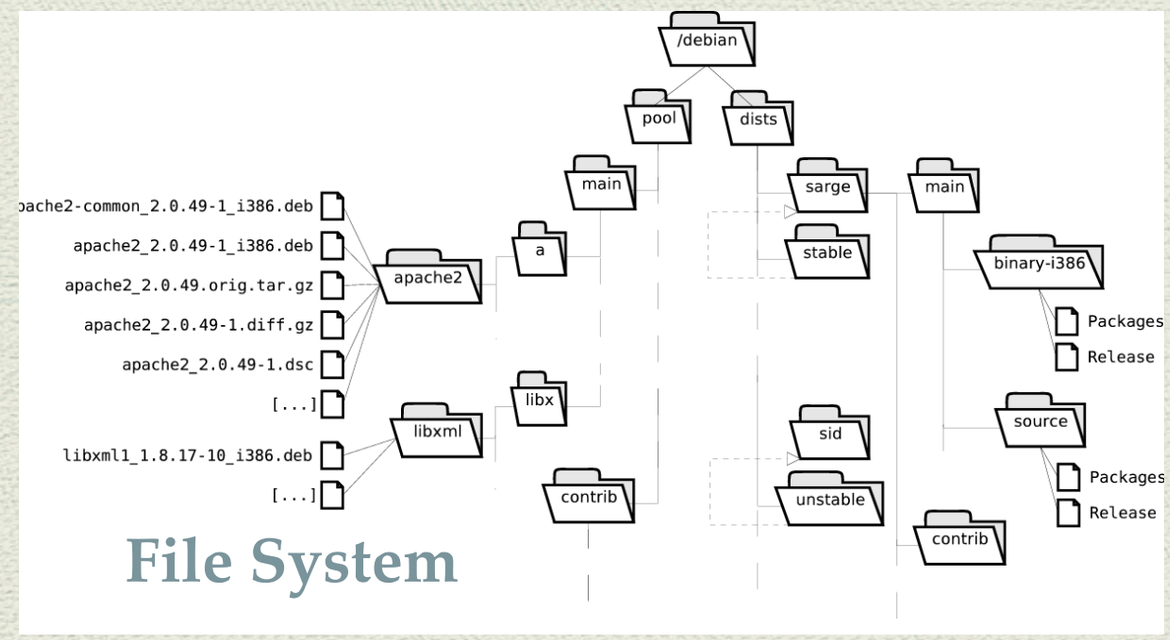
RICE

Hierarchical Data Structures

- Many applications use hierarchical data structures.

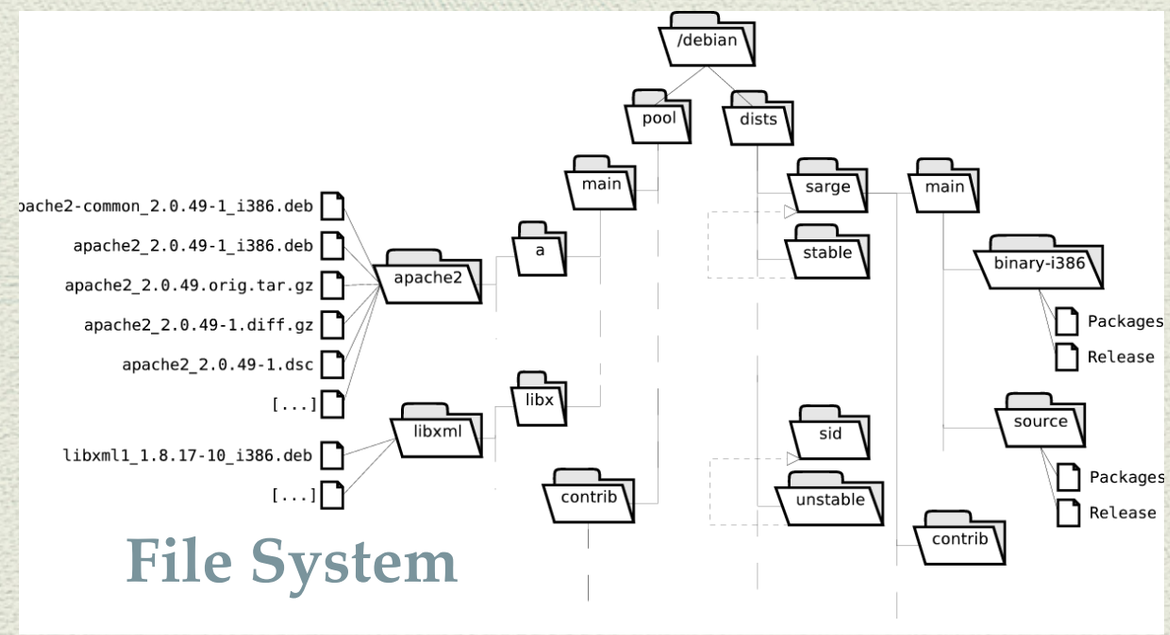
Hierarchical Data Structures

- Many applications use hierarchical data structures.



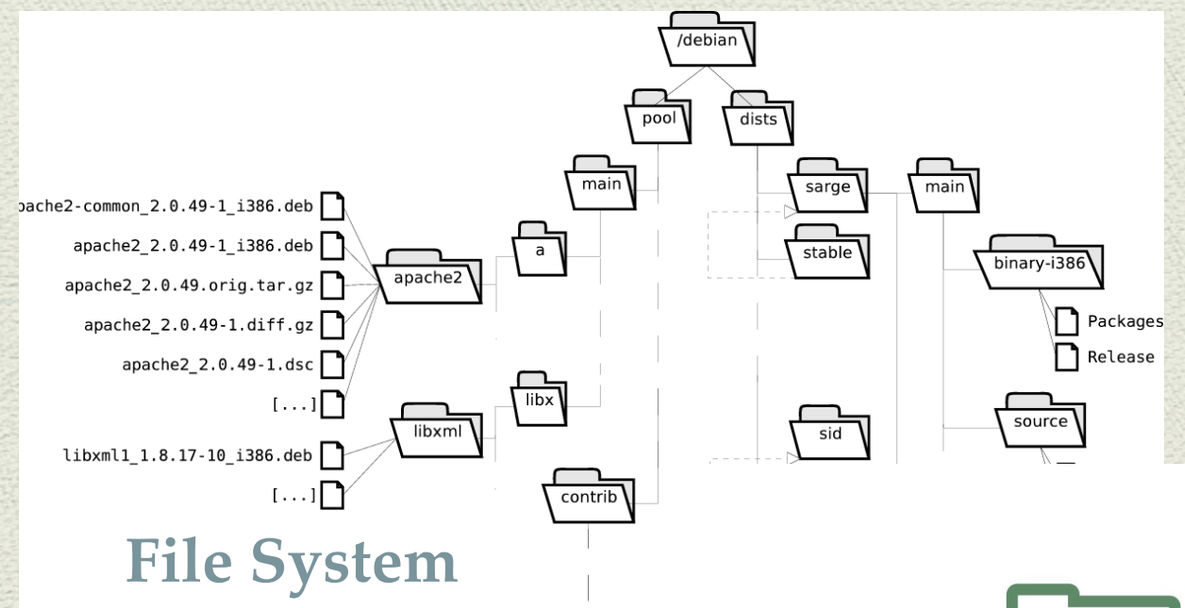
Hierarchical Data Structures

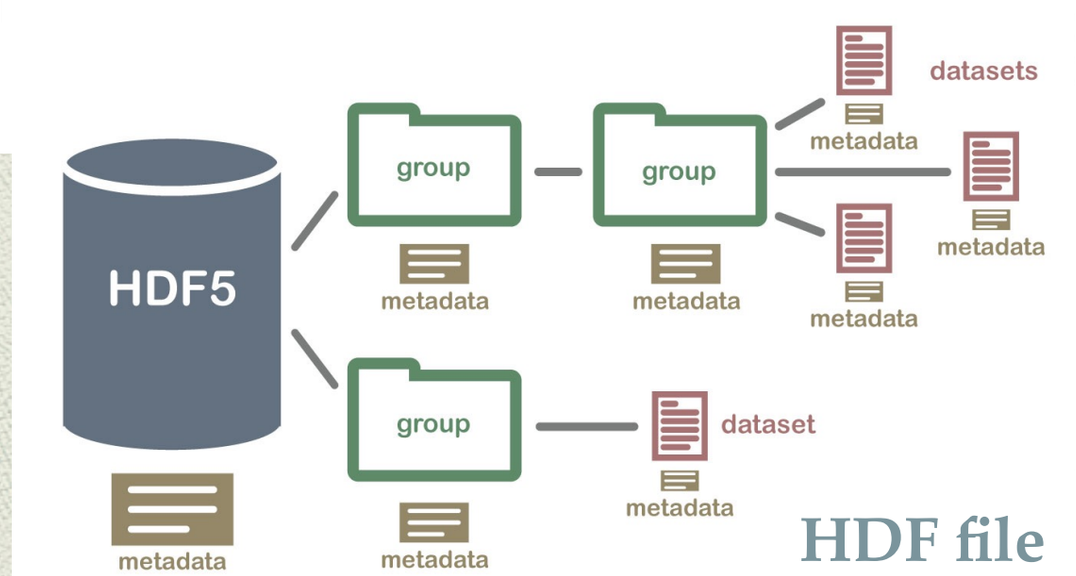
- Many applications use hierarchical data structures.



Hierarchical Data Structures

- Many applications use hierarchical data structures.

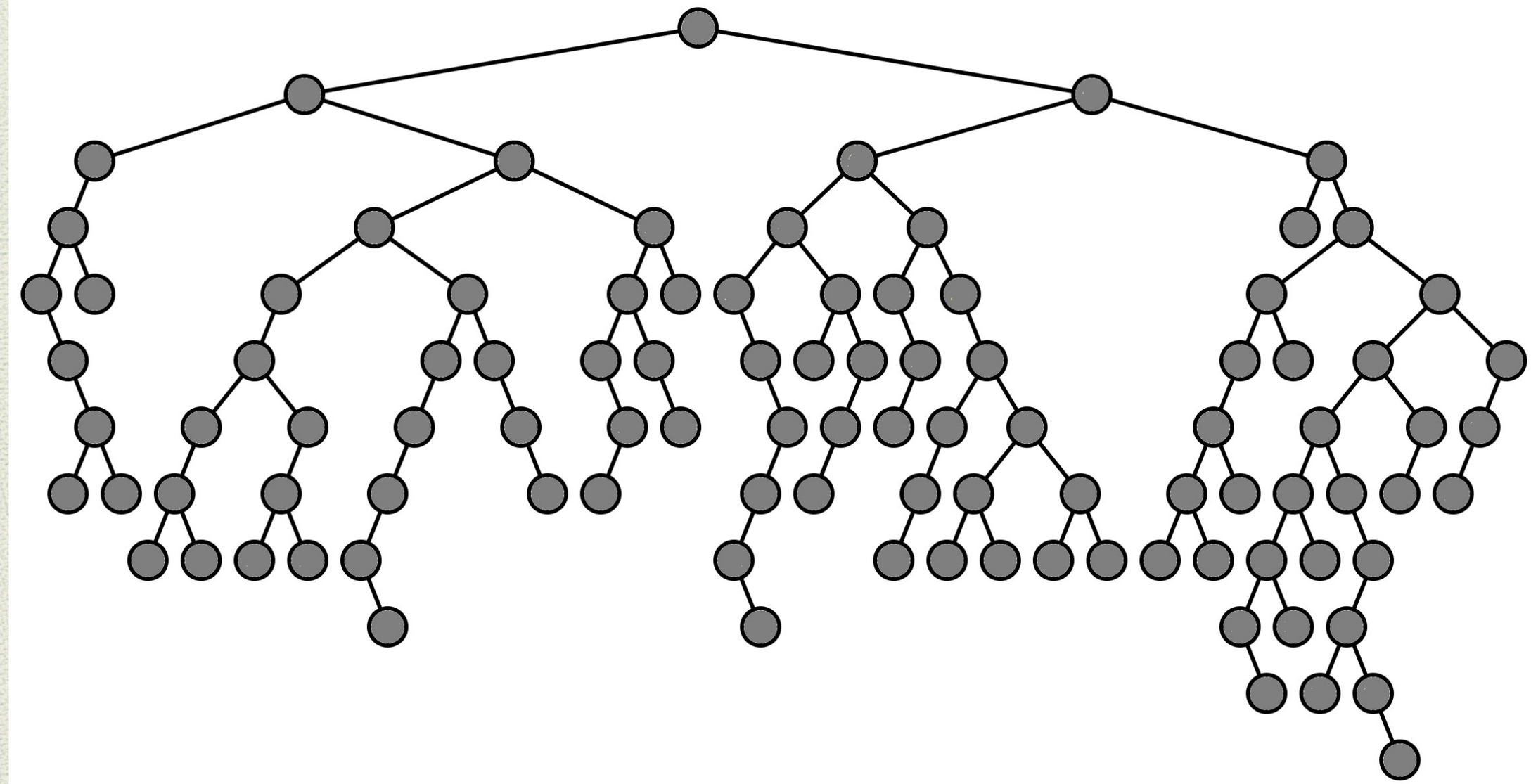




Hierarchical Data Structures

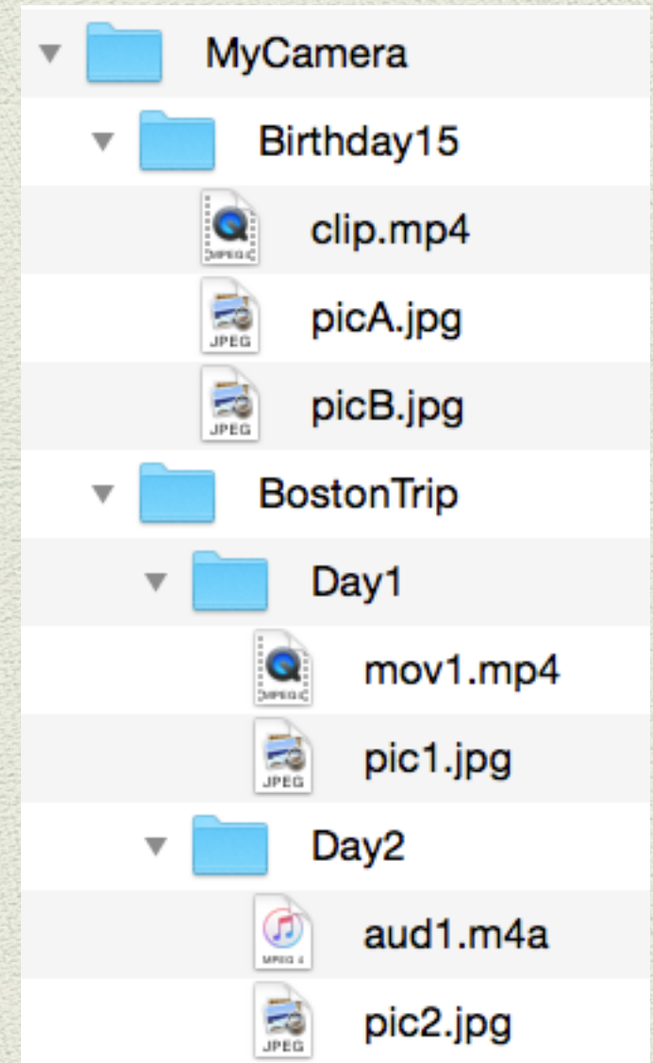
- Many applications use hierarchical data structures.

All of these structures are basically **Trees**.



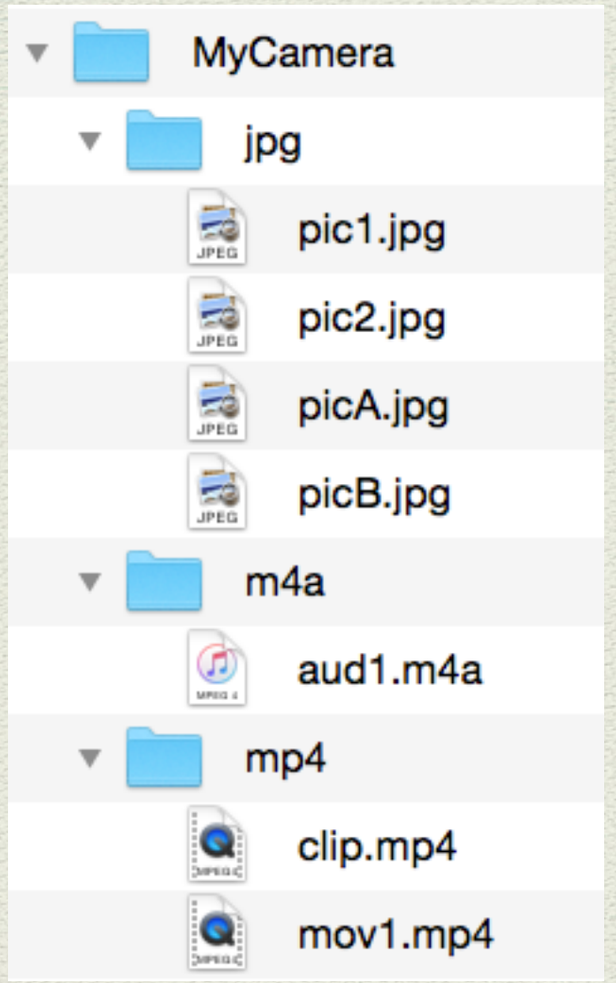
Tree Transformation

- Many user tasks require transforming the underlying tree.



Original Directory

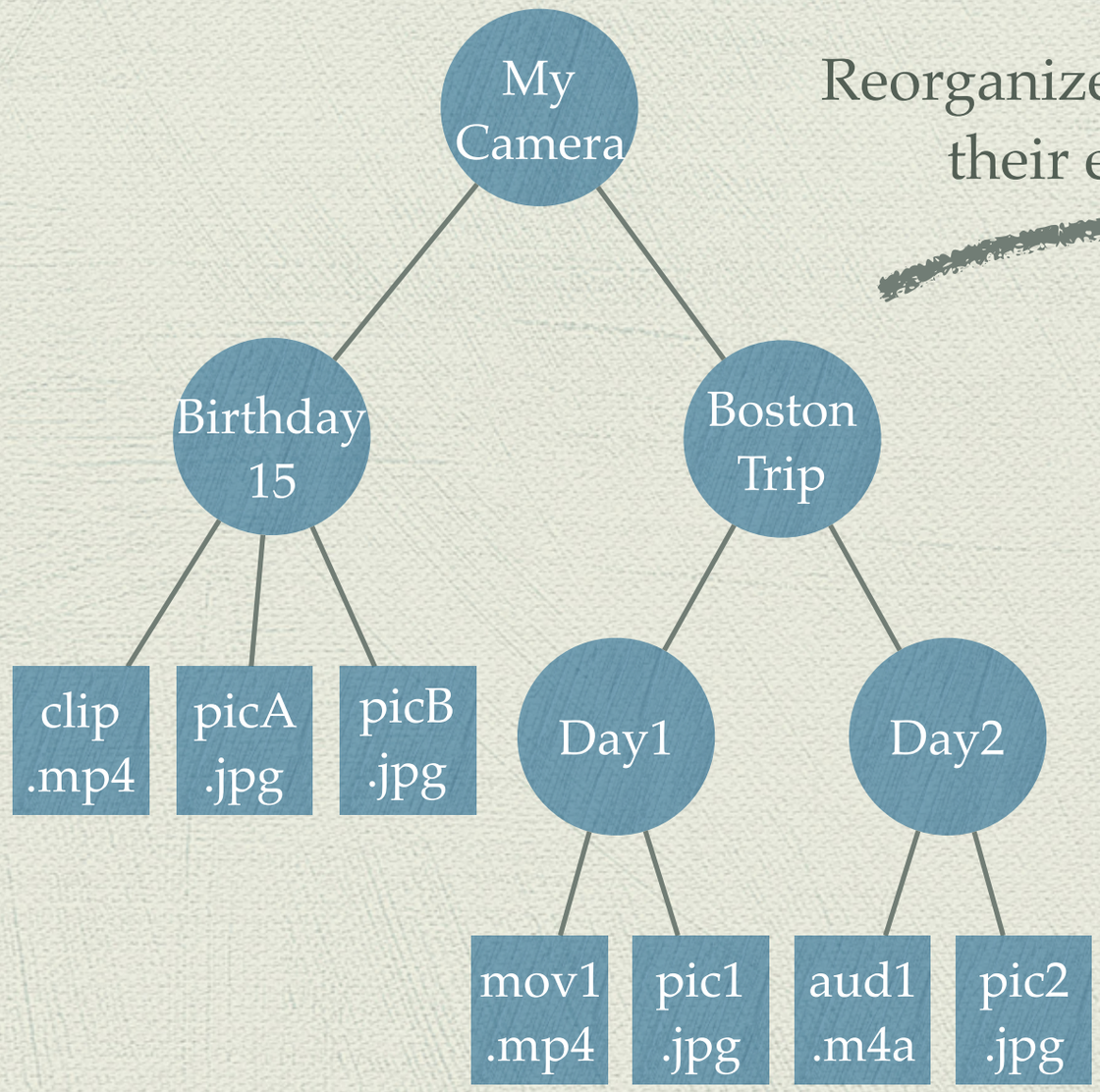
Reorganize files based on their extensions



Modified Directory

Tree Transformation

- Many user tasks require transforming the underlying tree.

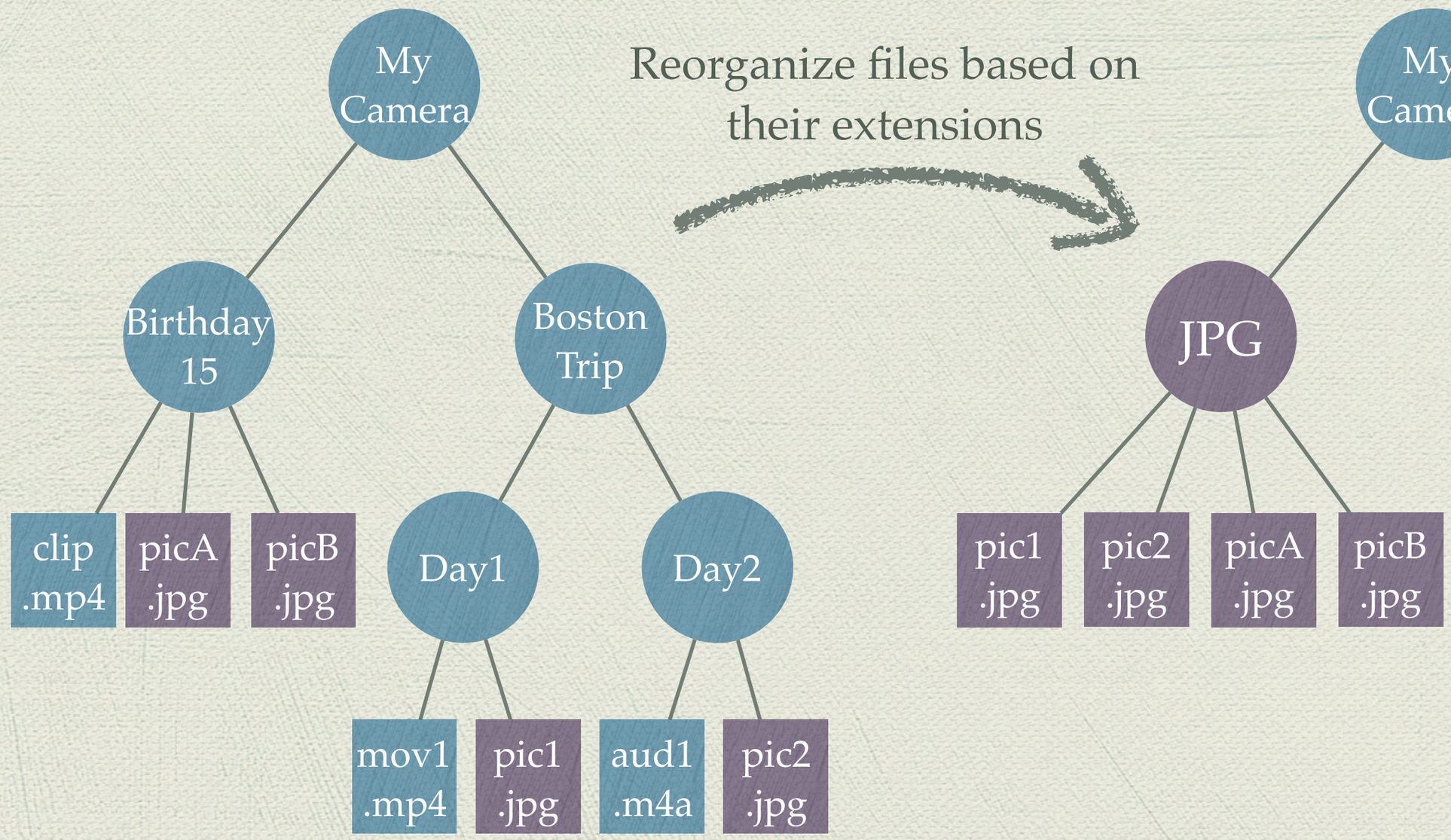


Original Tree

Reorganize files based on their extensions

Tree Transformation

- Many user tasks require transforming the underlying tree.

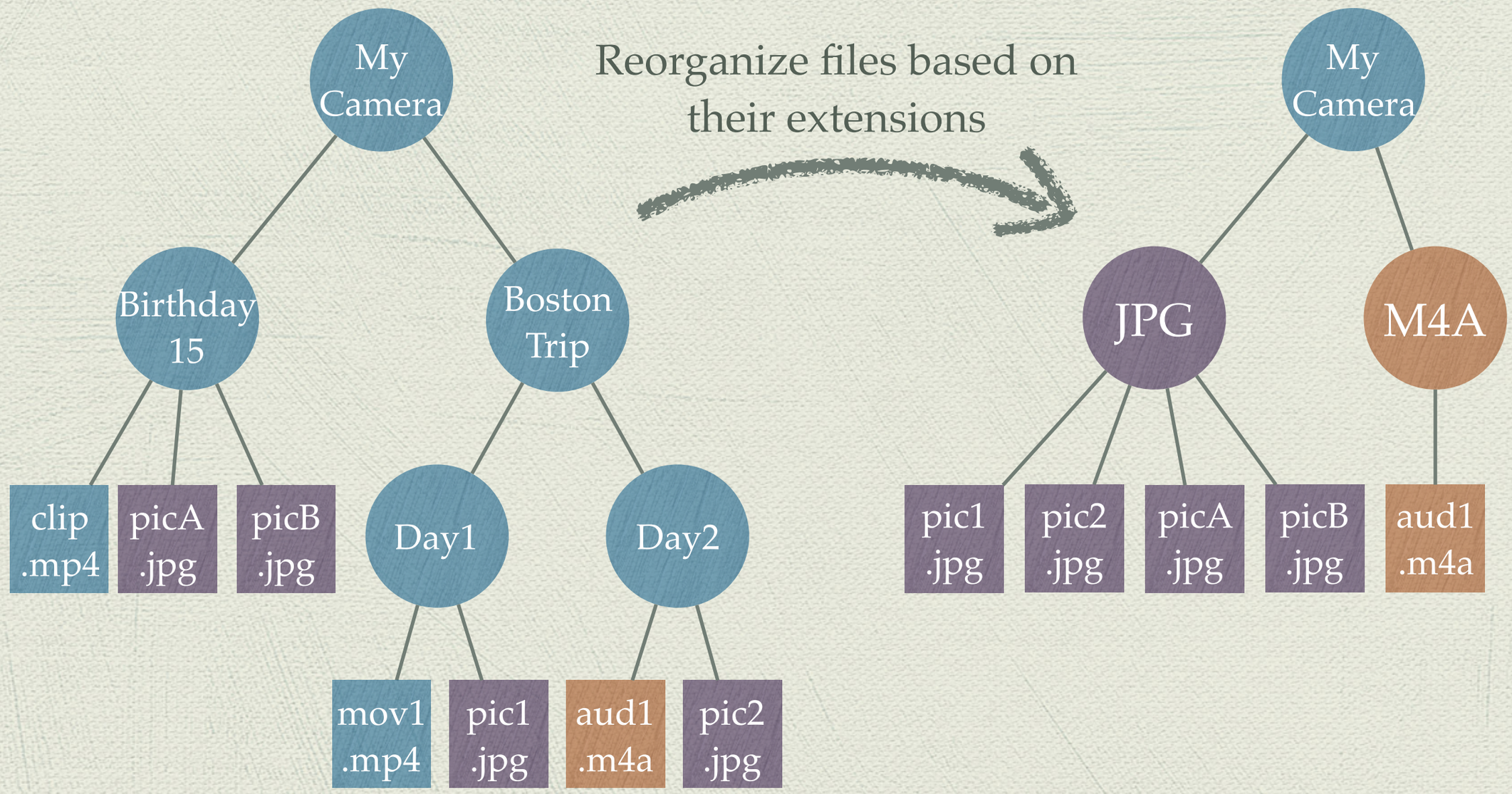


Original Tree

Modified Tree

Tree Transformation

- Many user tasks require transforming the underlying tree.

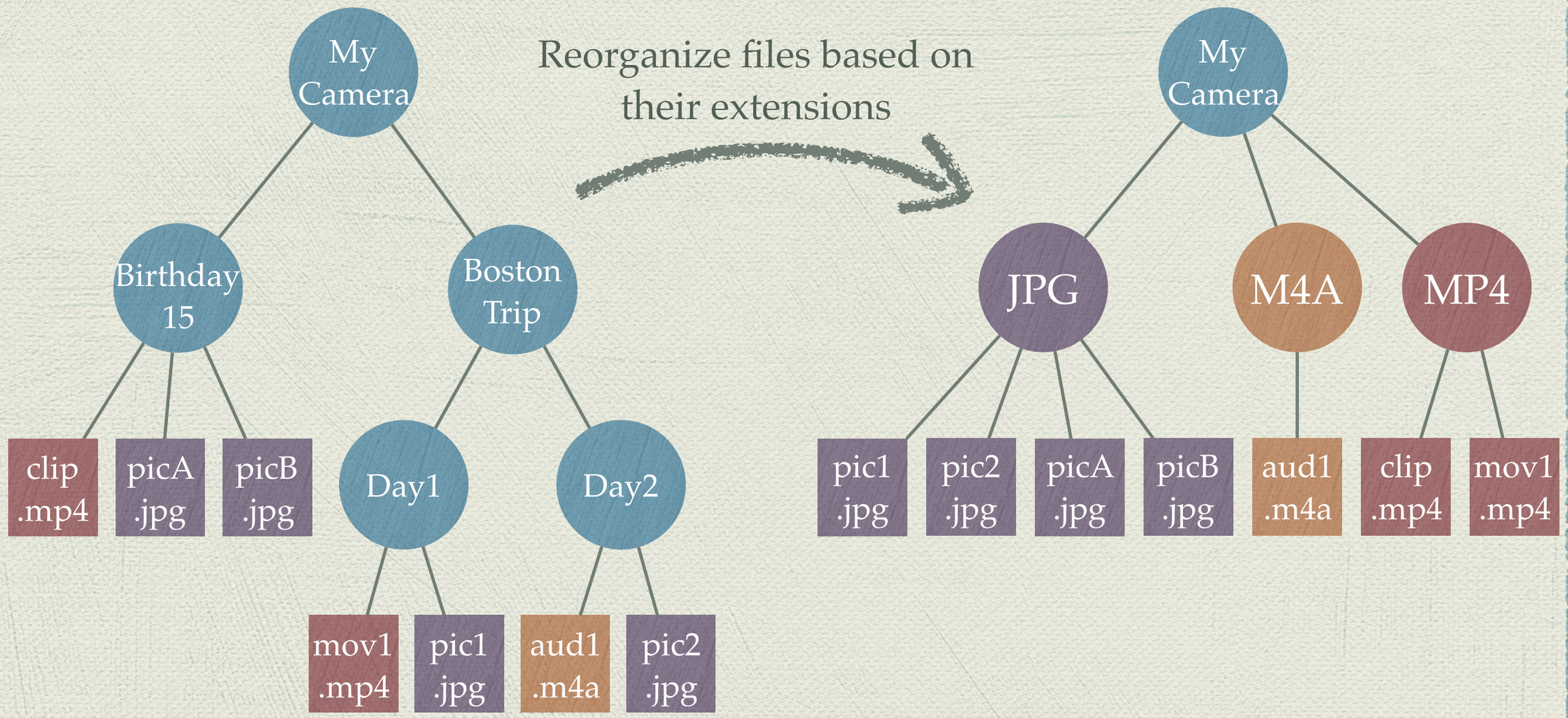


Original Tree

Modified Tree

Tree Transformation

- Many user tasks require transforming the underlying tree.



Original Tree

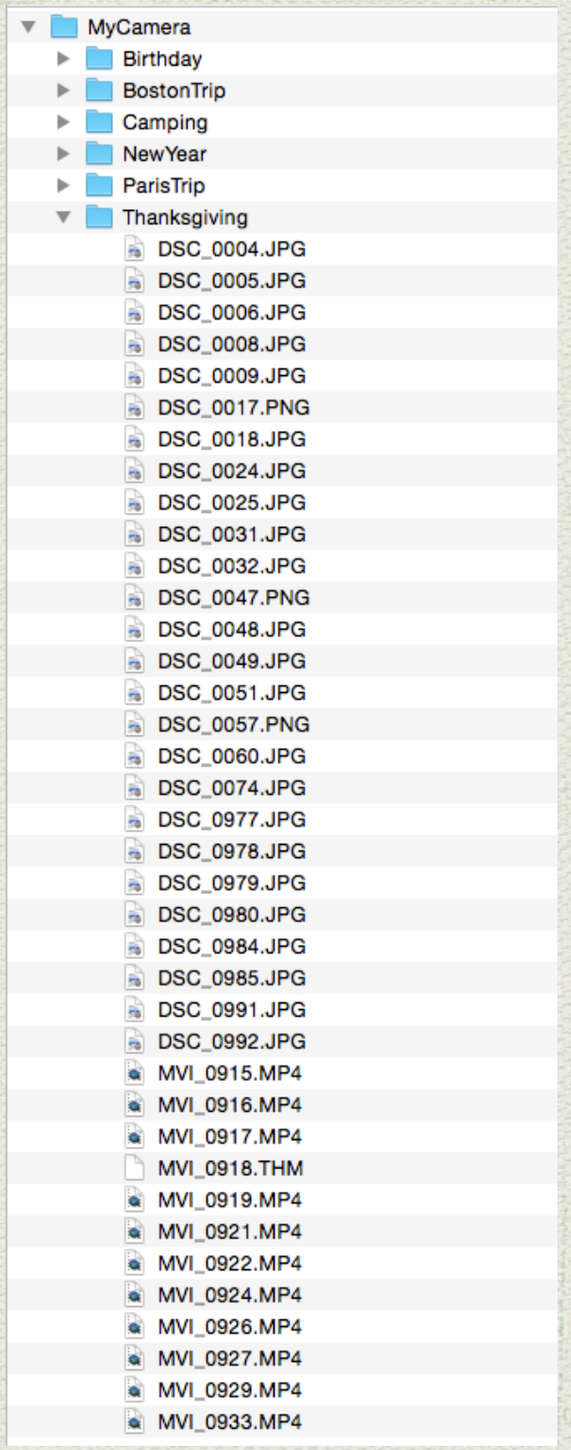
Modified Tree

Need for Synthesis

- Manual transformation is not feasible…

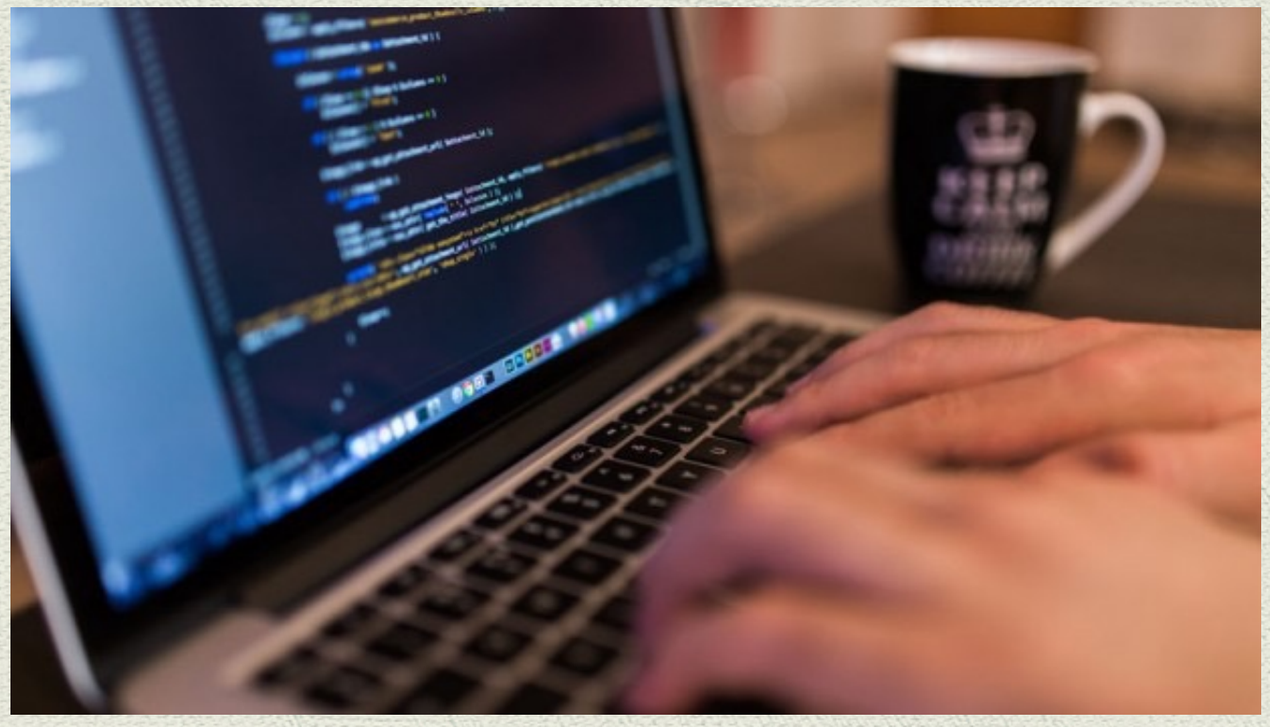
Need for Synthesis

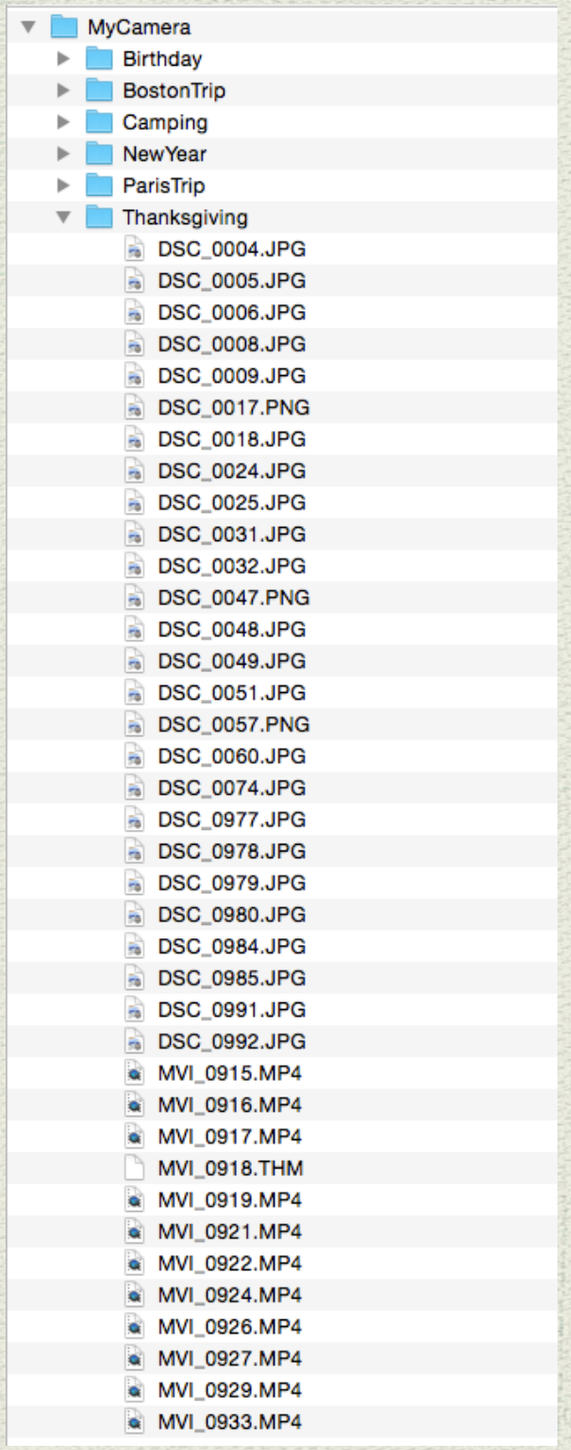
- Manual transformation is not feasible…



Need for Synthesis

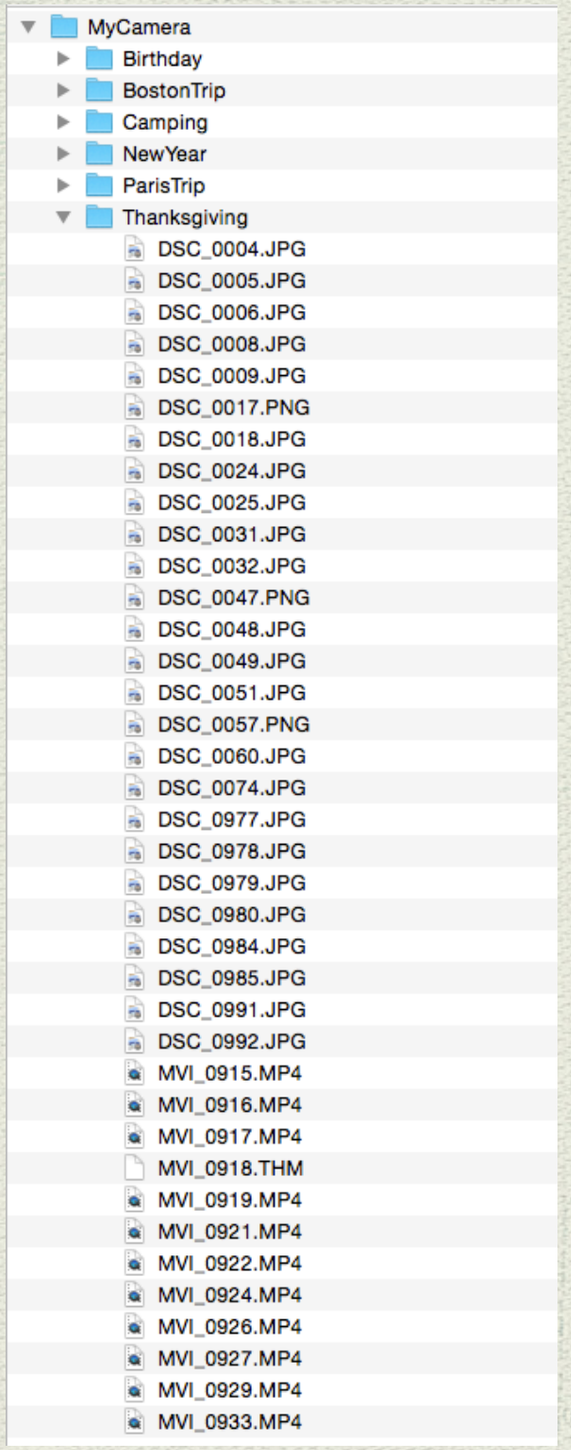
- Manual transformation is not feasible…
- So programmers write scripts.





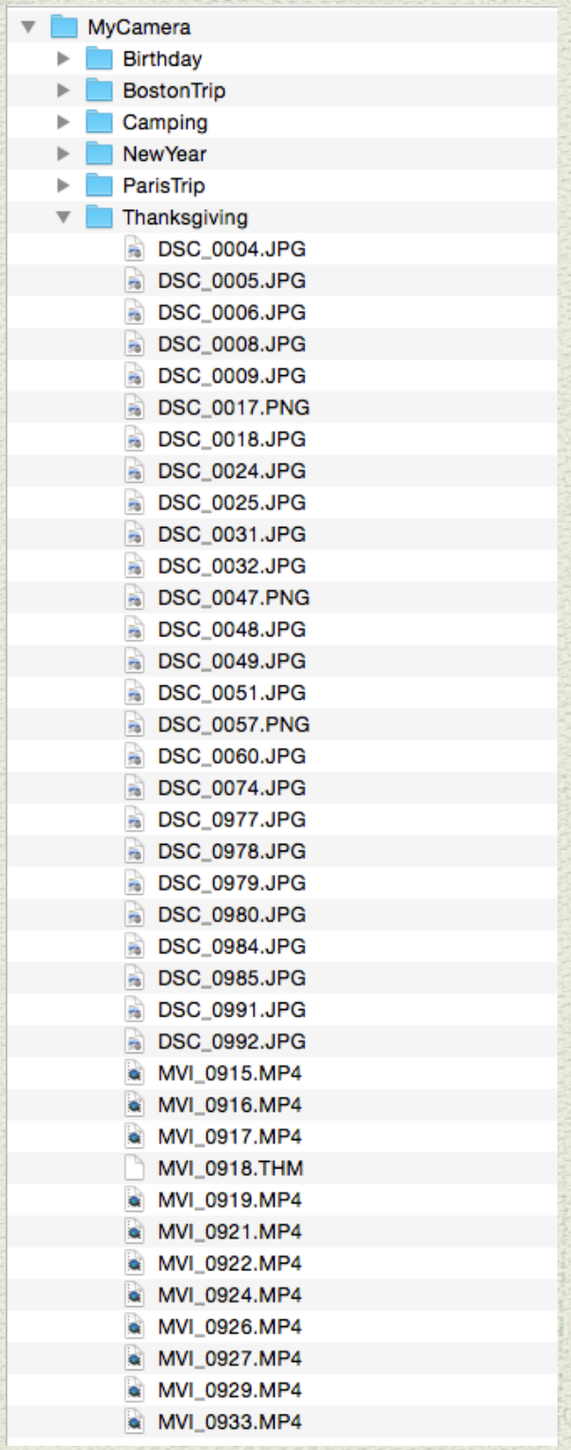
Need for Synthesis

- Manual transformation is not feasible…
- So programmers write scripts.
- But most users can't write scripts!



Need for Synthesis

- Manual transformation is not feasible…
- So programmers write scripts.
- But most users can't write scripts!
- **Solution:** Generate transformations automatically.

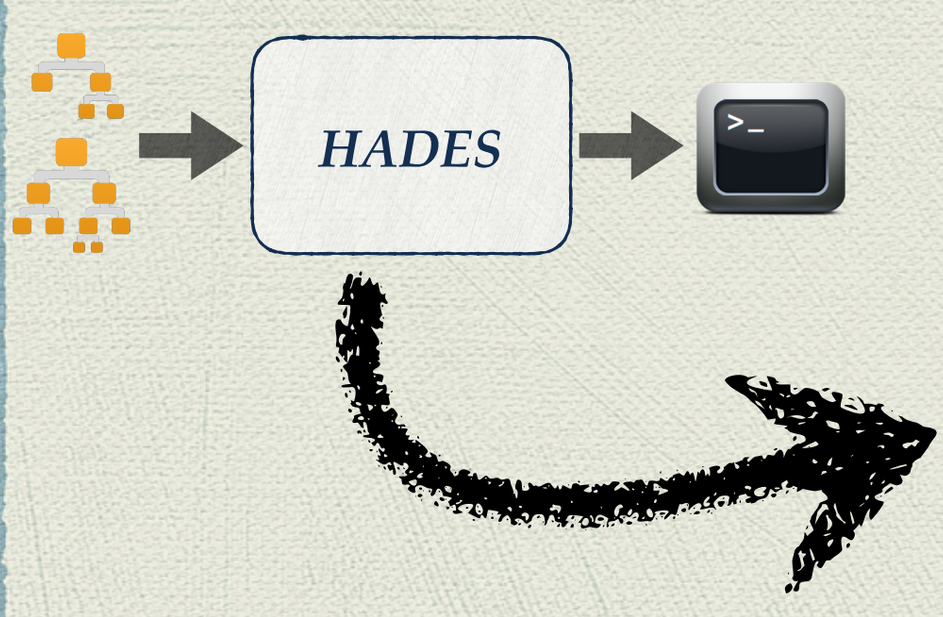


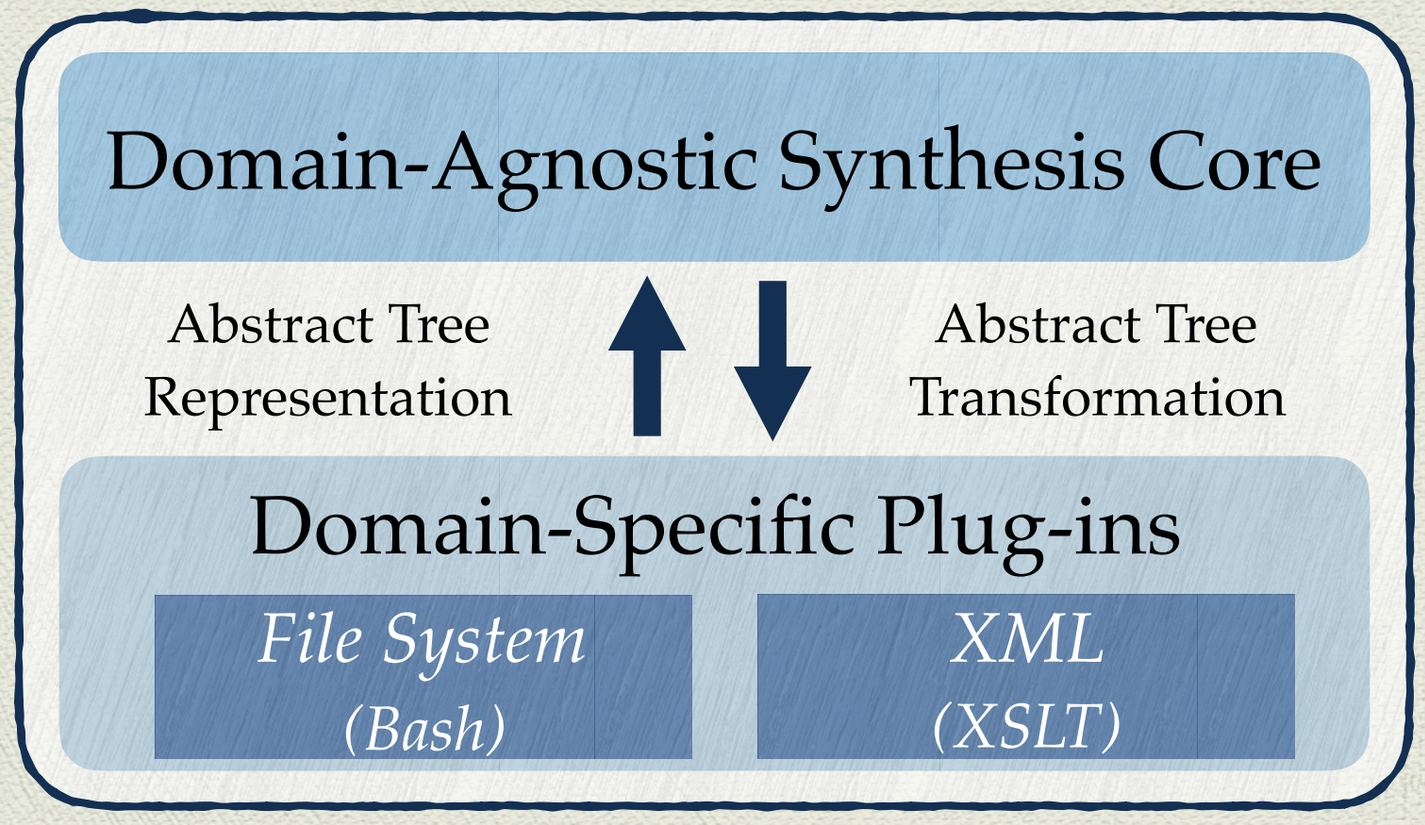
In This Talk…

- A new algorithm for automatically synthesizing tree transformations from input-output examples.

In This Talk…

- A new algorithm for automatically synthesizing tree transformations from input-output examples.
- Implementation of a domain-agnostic synthesis core and a set of domain-specific plug-ins for *File System* and *XML*.





State of the Art

- Much recent work on synthesizing programs on recursive data structures

State of the Art

- Much recent work on synthesizing programs on recursive data structures
  - λ^2 [Feser et. al.], Myth [Oseraa& Zdancewic], …
  - Synthesize higher-order functional programs

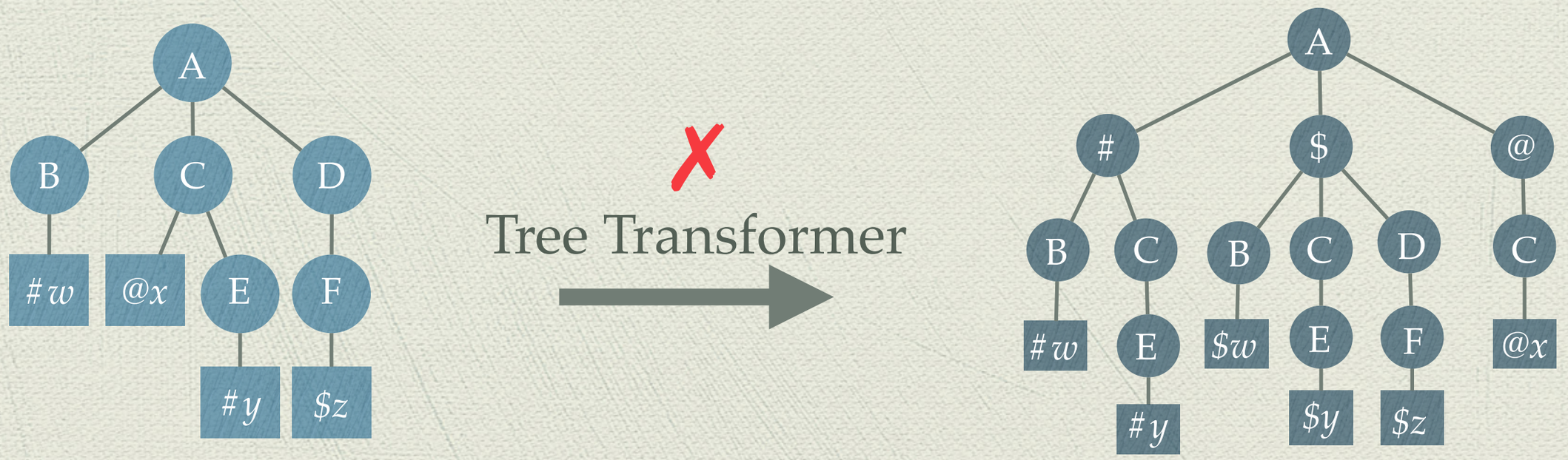
State of the Art

- Much recent work on synthesizing programs on recursive data structures
  - λ^2 [Feser et. al.], Myth [Oseraa& Zdancewic], …
  - Synthesize higher-order functional programs

State of the Art

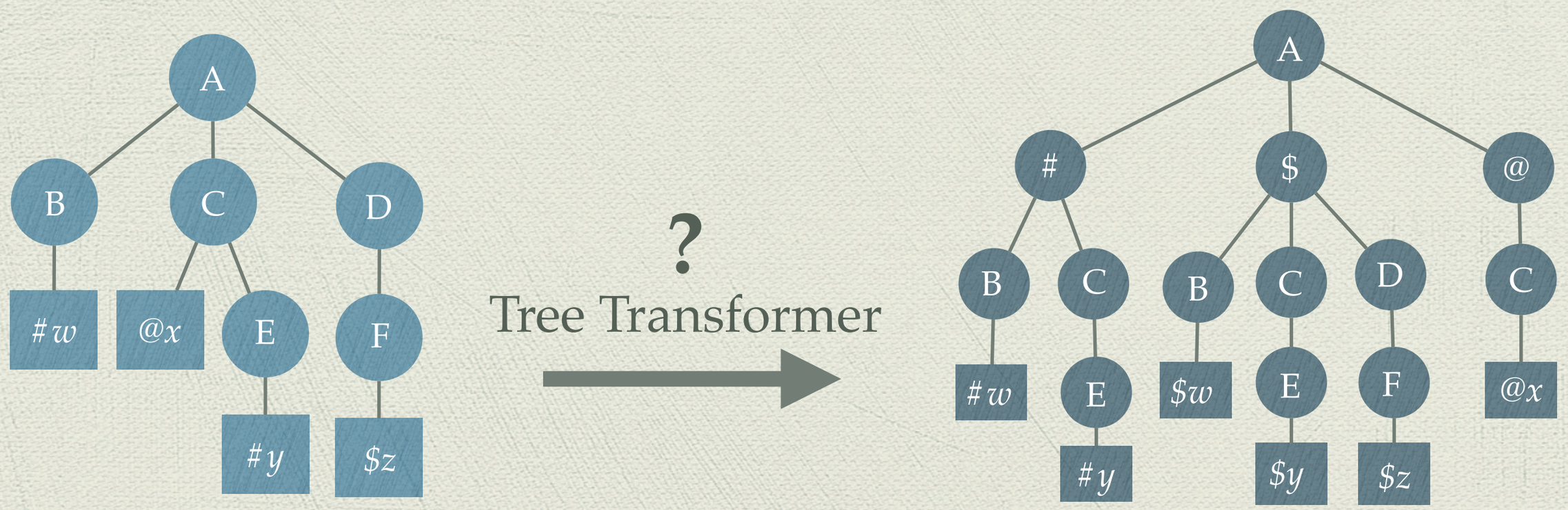
- Much recent work on synthesizing programs on recursive data structures
  - λ^2 [Feser et. al.], Myth [Oseraa& Zdancewic], …
  - Synthesize higher-order functional programs

- Don't support many tree transformations (e.g. transformation which alters the tree structure)
  - Hard to implement using functional approach



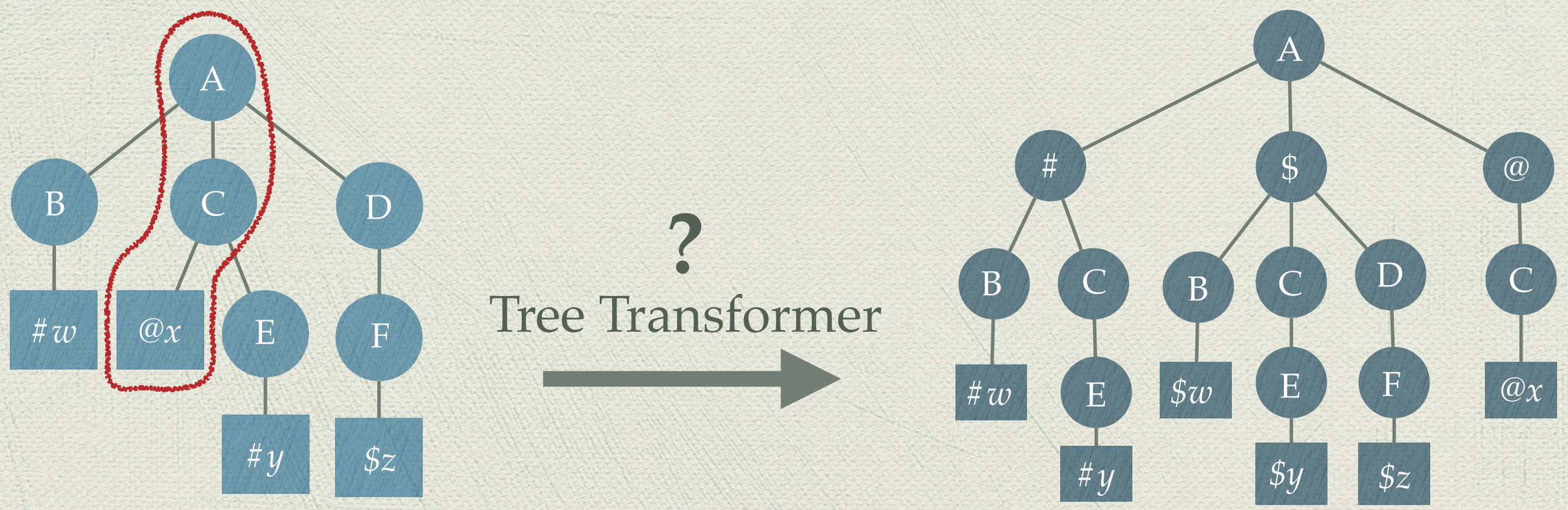
Key Idea

- Different way to look at tree transformation:



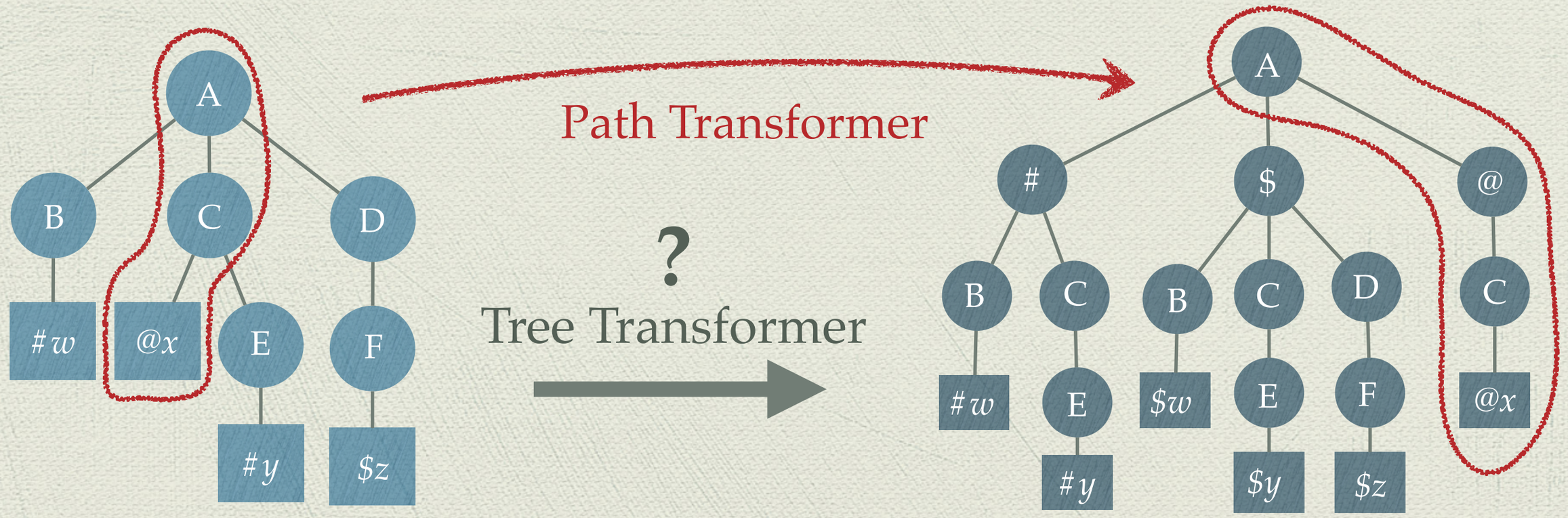
Key Idea

- Different way to look at tree transformation:
  - Think of a tree as a set of paths.



Key Idea

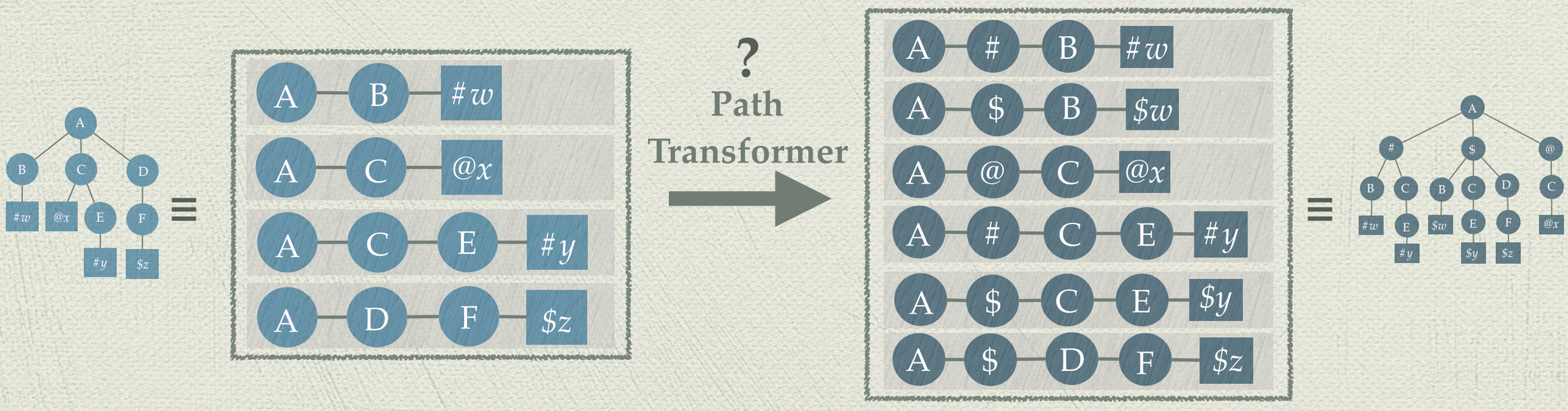
- Different way to look at tree transformation:
  - Think of a tree as a set of paths.
  - Synthesize a transformer which transforms a path ***P*** to path ***P′***.



Key Idea

- Different way to look at tree transformation:
  - Think of a tree as a set of paths.
  - Synthesize a transformer which transforms a path ***P*** to path ***P′***.

> Reduce tree transformation synthesis to synthesis of a transformer over paths of the tree.



This Approach…

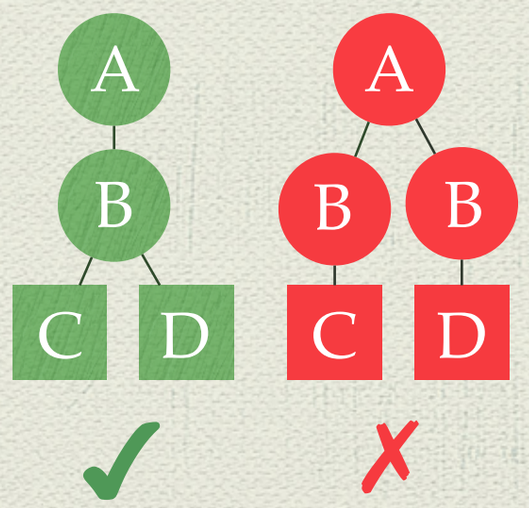
- Simplifies the synthesis task.

This Approach…

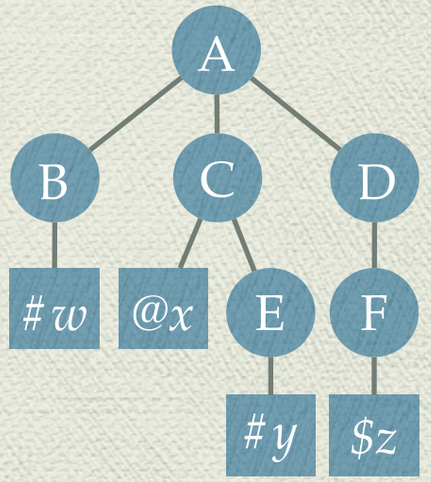
- Simplifies the synthesis task.
- Handles a broad class of tree transformations.

This Approach…

- Simplifies the synthesis task.
- Handles a broad class of tree transformations.
- Relies on a natural assumption:
  - Trees do not have siblings with the same label.
  - Many applications (e.g. File System) preserve this.



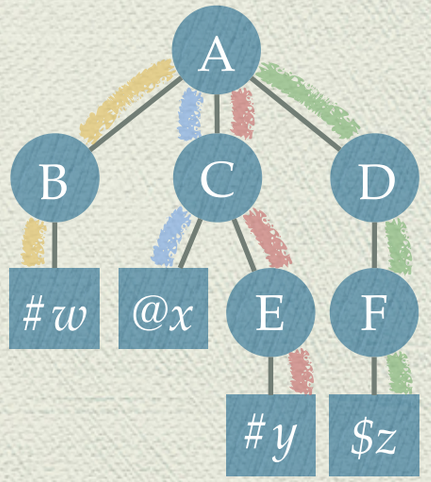
Tree Transformer



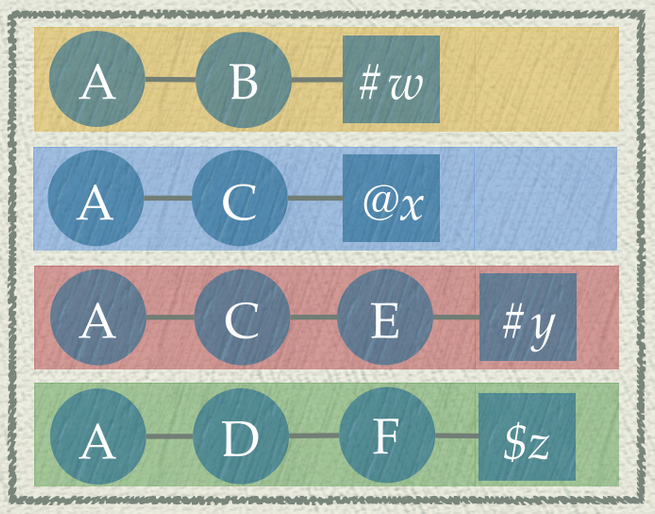
Tree Transformation

```
TreeTrans(T)
{




  return T';
}
```

Tree Transformer



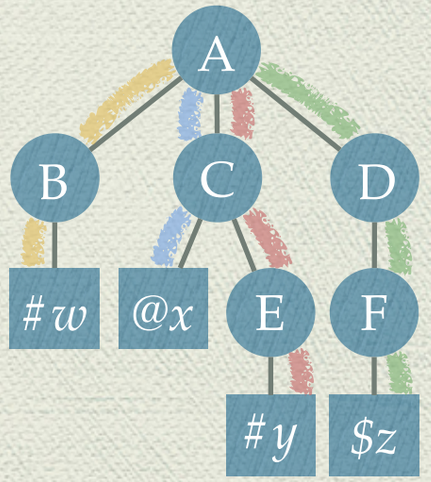
Furcation



Tree Transformation

```
TreeTrans(T)
{
  P = furcate(T);


  return T';
}
```

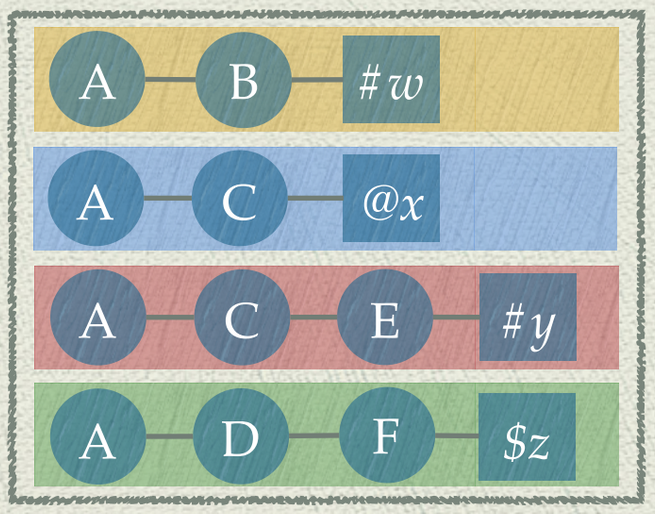
Tree Transformer



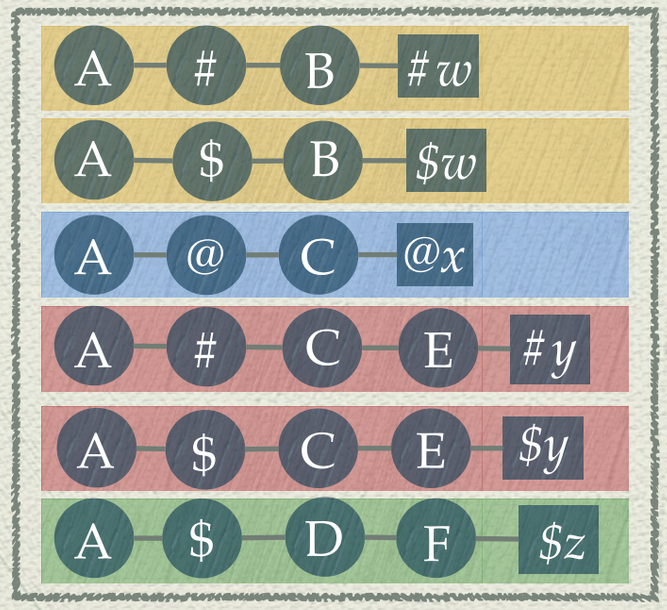
Tree Transformation

```
TreeTrans(T)
{
  P = furcate(T);
  foreach p ∈ P
    P′ = P′ ∪ pathTrans(p);

  return T′;
}
```

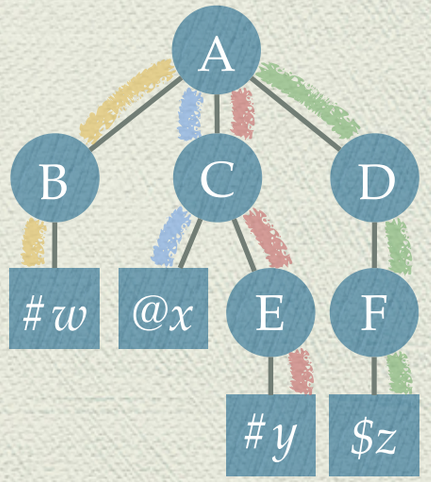
Furcation



Path Transformation



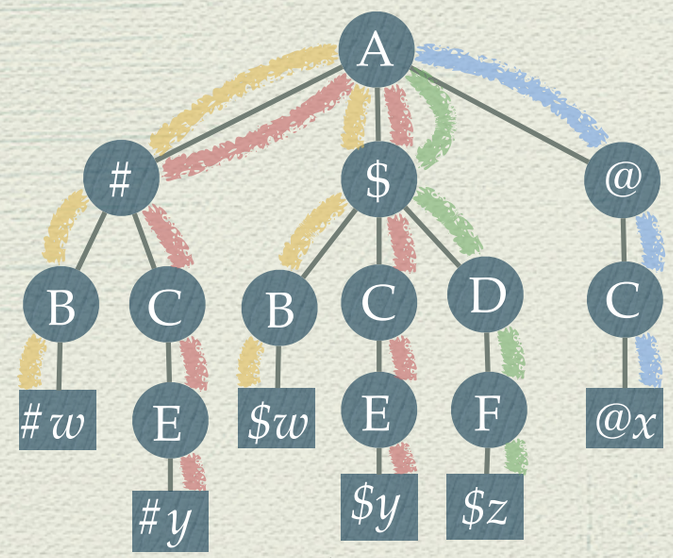
Tree Transformer



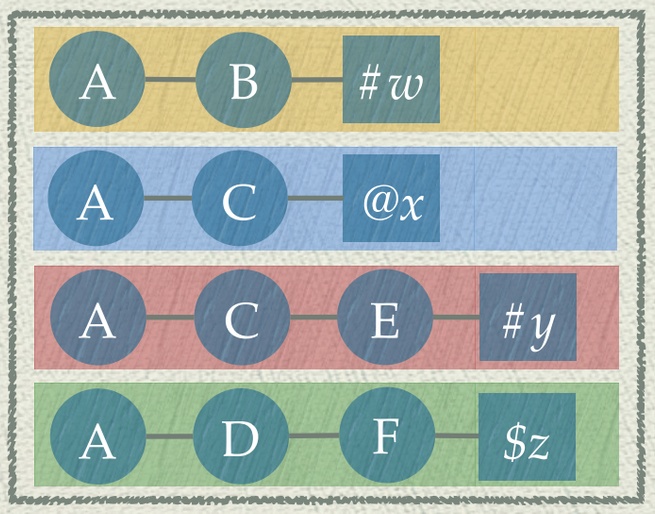
Tree Transformation

Furcation

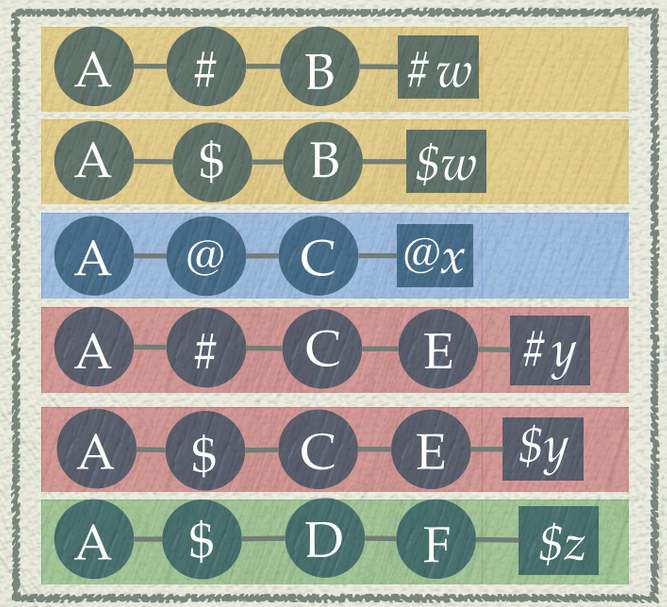
```
TreeTrans(T)
{
  P = furcate(T);
  foreach p ∈ P
    P′ = P′ ∪ pathTrans(p);
  T′ = splice(P′);
  return T′;
}
```



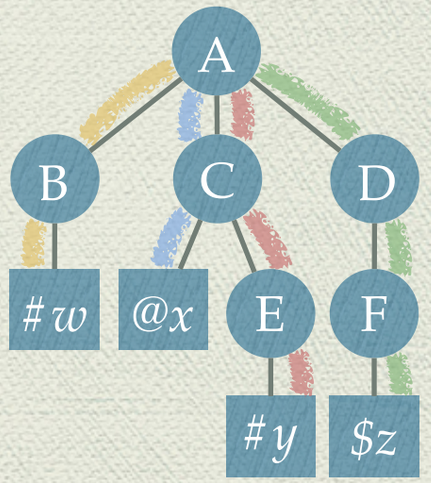
Splicing



Path Transformation



Tree Transformer

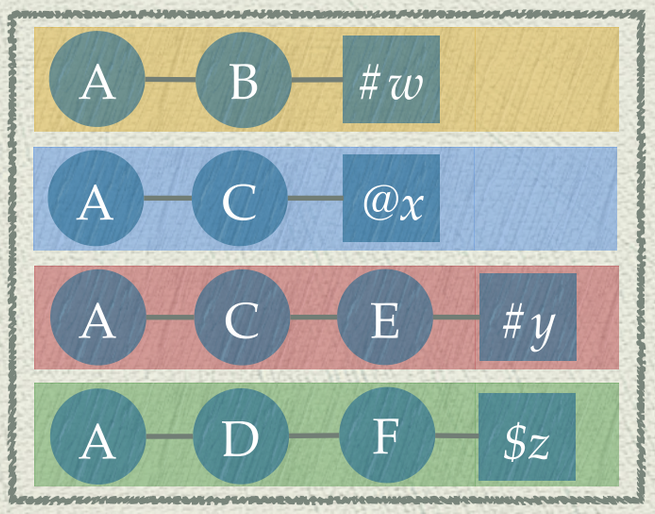


Tree Transformation

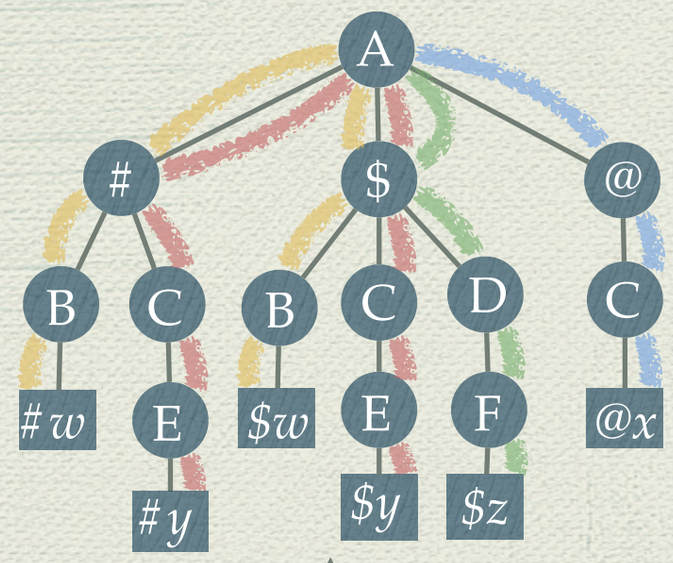
Furcation

Known

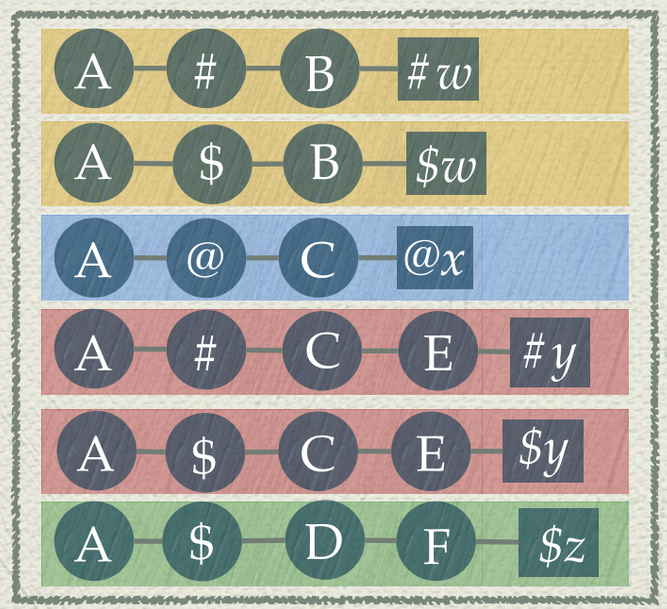
```
TreeTrans(T)
 {
  P = furcate(T);
  foreach p ∈ P
    P′ = P′ ∪ pathTrans(p);
  T′ = splice(P′);
  return T′;
}
```



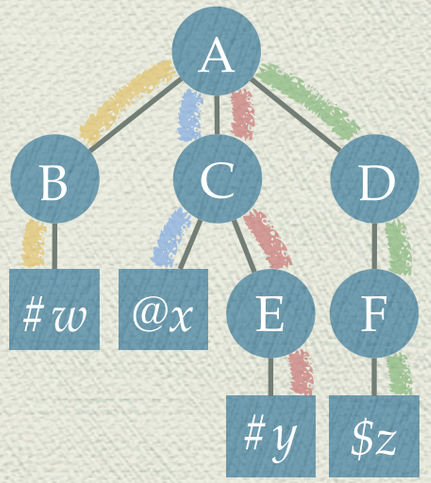
Path Transformation



Splicing



Tree Transformer



Tree Transformation

Furcation

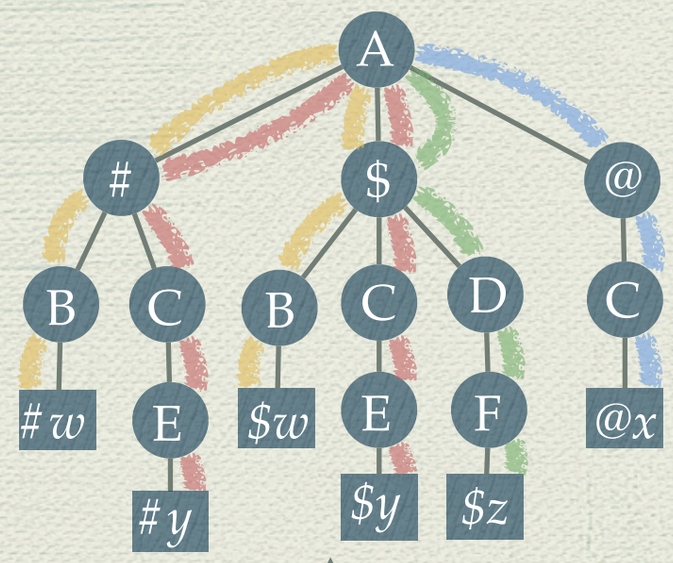
Known

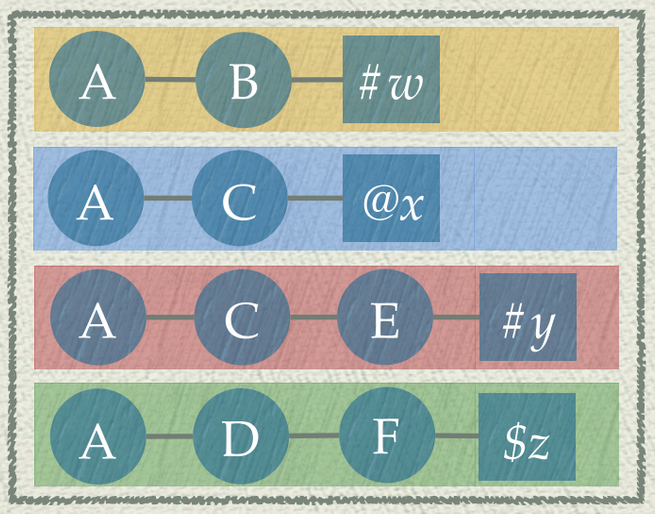
```
TreeTrans(T)
 {
  P = furcate(T);
  foreach p ∈ P
    P′ = P′ ∪ pathTrans(p);
  T′ = splice(P′);
  return T′;
}
```

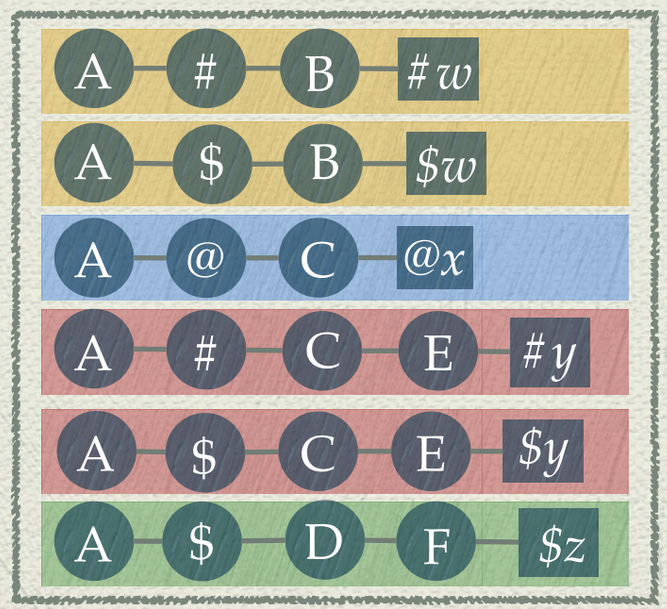
To be synthesized

Splicing

Path Transformation

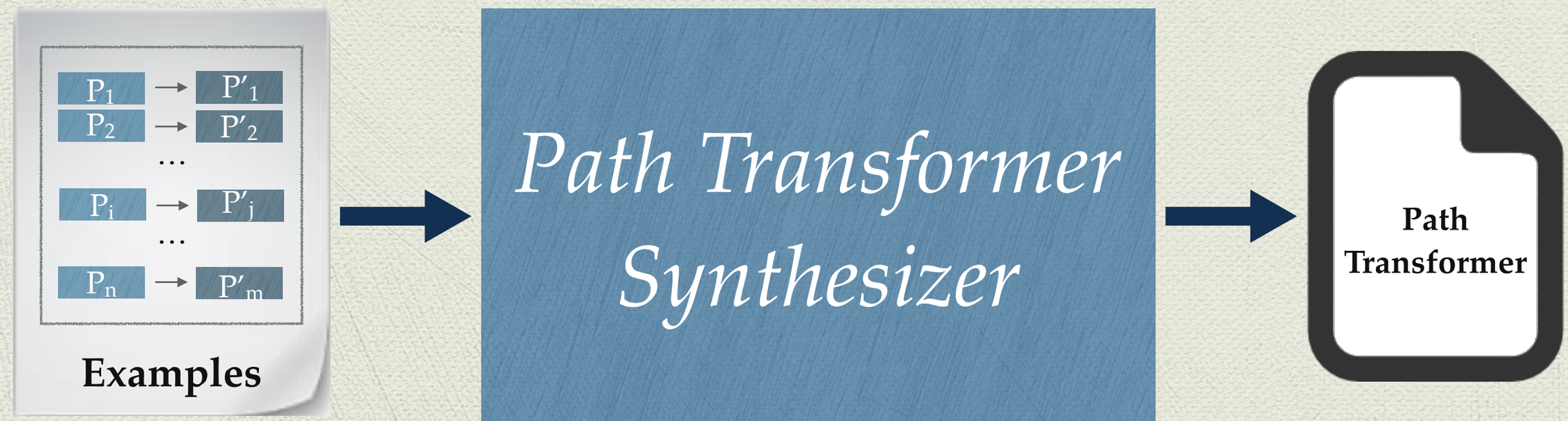






Path Transformation Synthesis

- New algorithm for path transformation synthesis based on:
  - **Enumerative search**
  - **SMT solving**
  - **Decision tree learning**



Path Transformer Language

Need simple language that achieves good tradeoff between expressiveness and ease of synthesis

PathTransformer (p)

Path Transformer Language

Need simple language that achieves good tradeoff between expressiveness and ease of synthesis

- Set of conditional statements:
  if ($C_i(p)$) then $U_i(p)$

PathTransformer (p)

$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$

Path Transformer Language

Need simple language that achieves good tradeoff between expressiveness and ease of synthesis

- Set of conditional statements:
  if ($C_i(p)$) then $U_i(p)$
- Predicate C_i:
  Check properties on input path
- Unifier U_i:
  Conditional free transformer function

PathTransformer (p)

$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$

Path Transformer Language

Need simple language that achieves good tradeoff between expressiveness and ease of synthesis

- Set of conditional statements:
  if ($C_i(p)$) then $U_i(p)$
- Predicate C_i:
  Check properties on input path
- Unifier U_i:
  Conditional free transformer function

PathTransformer (p)

$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

$$U_i(\mathrm{p}) = \mathrm{concat}\,(g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))$$

$$g_j(\mathrm{p}) = \mathrm{map}\; F_j\; \mathrm{subpath}(\mathrm{p}, t_j, t'_j)$$

Drawn from a finite vocabulary (e.g. doc2pdf, chmod, zip, …)

start index

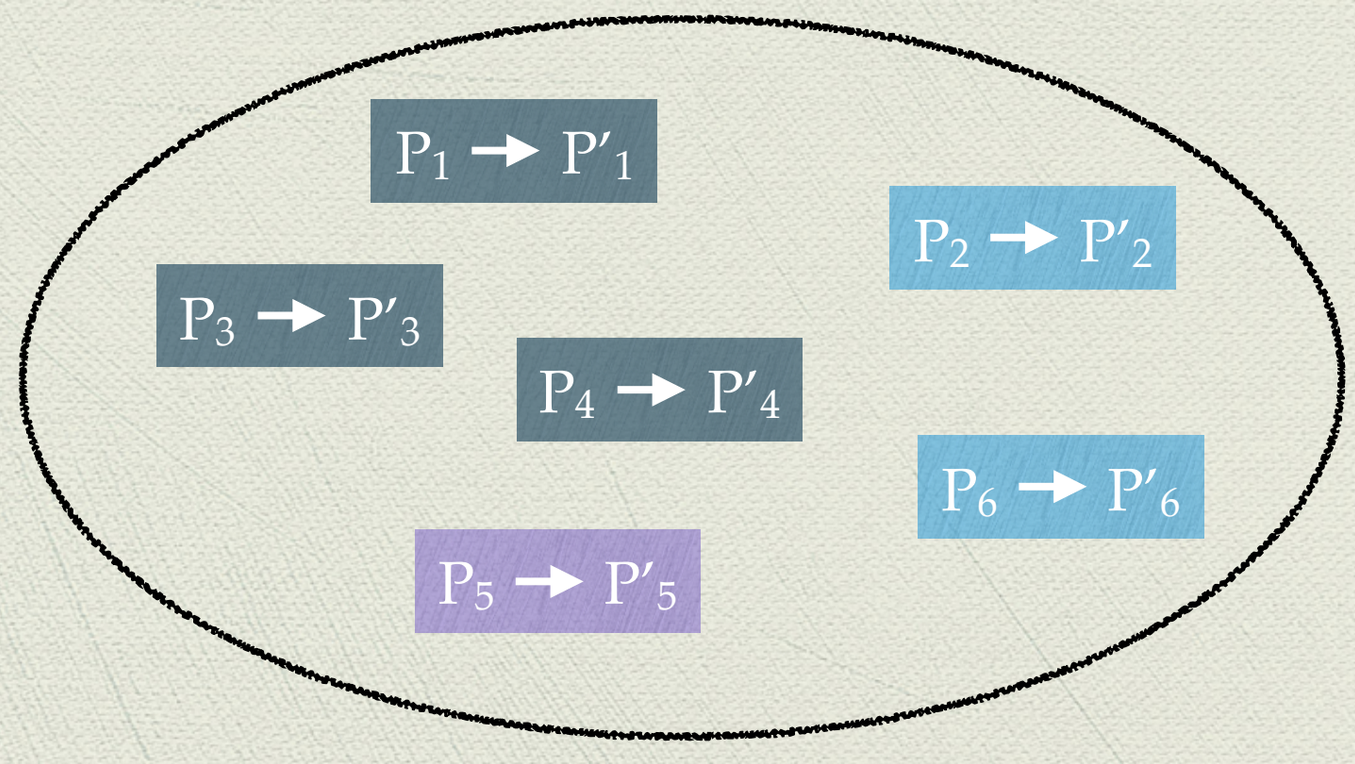
end index

Partition

$$\{C_1(\mathbf{p}) \rightarrow U_1(\mathbf{p}) \; ; \ldots ; \; C_n(\mathbf{p}) \rightarrow U_n(\mathbf{p})\}$$

$$U_i(\mathrm{p}) = \mathrm{concat}\,(g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))$$

$$g_j(\mathrm{p}) = \mathrm{map}\; F_j\; \mathrm{subpath}(\mathrm{p}, \mathrm{t_j}, \mathrm{t'_j})$$

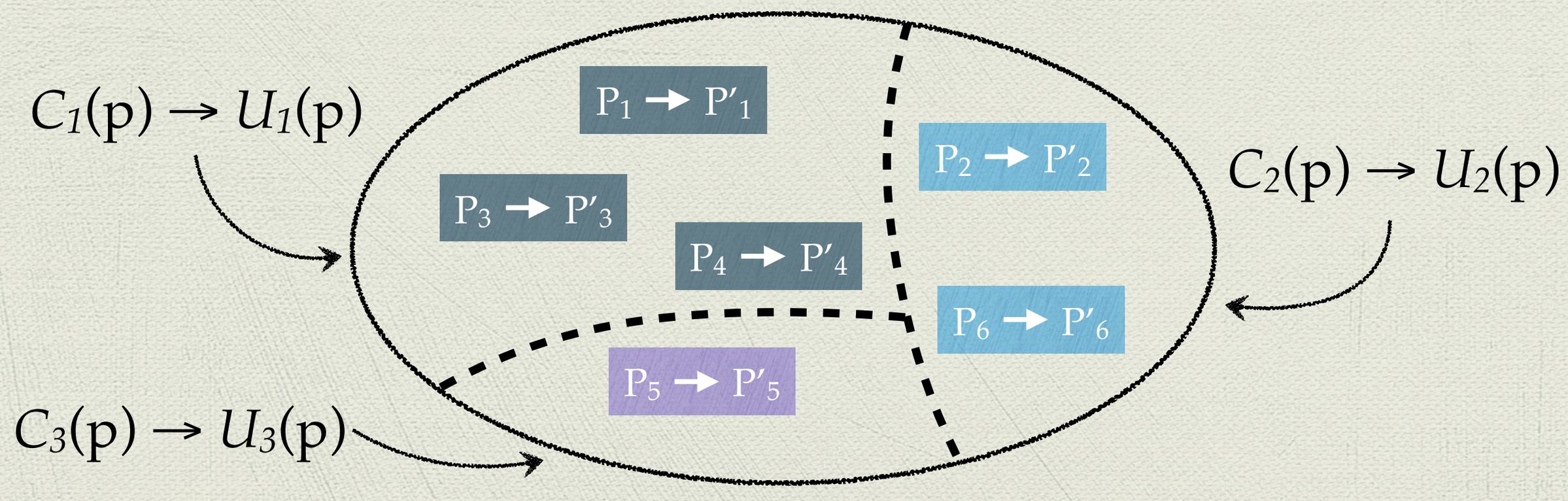


Partition

$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

$$U_i(\mathrm{p}) = \mathrm{concat}\,(g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))$$

$$g_j(\mathrm{p}) = \mathrm{map}\; F_j\; \mathrm{subpath}(\mathrm{p}, t_j, t'_j)$$



Partition

$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

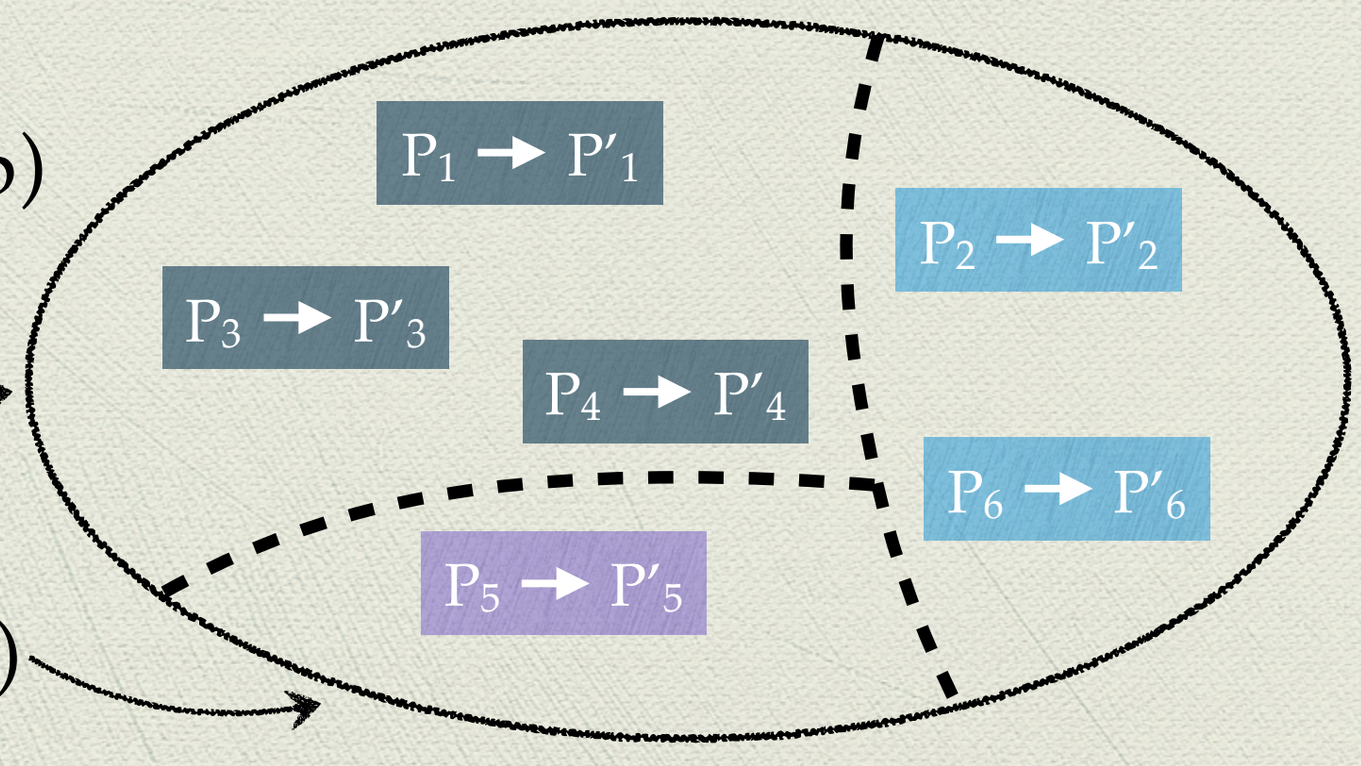
$$U_i(\mathrm{p}) = \mathrm{concat}\,(g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))$$

$$g_j(\mathrm{p}) = \mathrm{map}\; F_j\; \mathrm{subpath}(\mathrm{p}, t_j, t'_j)$$

Partition 1
$C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p})$

Partition 2
$C_2(\mathrm{p}) \rightarrow U_2(\mathrm{p})$

Partition 3
$C_3(\mathrm{p}) \rightarrow U_3(\mathrm{p})$



Unifier

$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

$$\boldsymbol{U_i(\mathrm{p}) = \mathbf{concat}\ (g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))}$$

$$\boldsymbol{g_j(\mathrm{p}) = \mathbf{map}\ F_j\ \mathbf{subpath}(\mathrm{p}, t_j, t'_j)}$$

Input Path

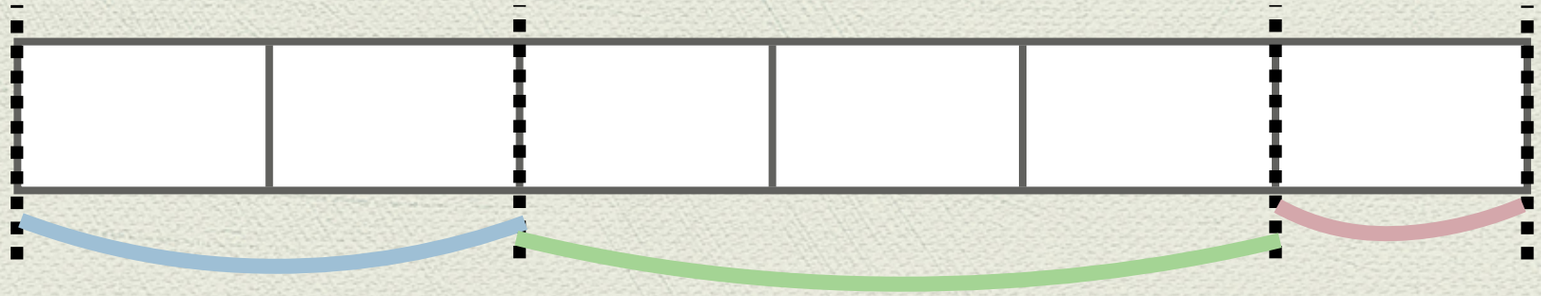
Unifier

$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

$$\boldsymbol{U_i(\mathrm{p}) = \mathbf{concat}\ (g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))}$$

$$\boldsymbol{g_j(\mathrm{p}) = \mathbf{map}\ F_j\ \mathbf{subpath}(\mathrm{p}, t_j, t'_j)}$$

Input Path



Unifier

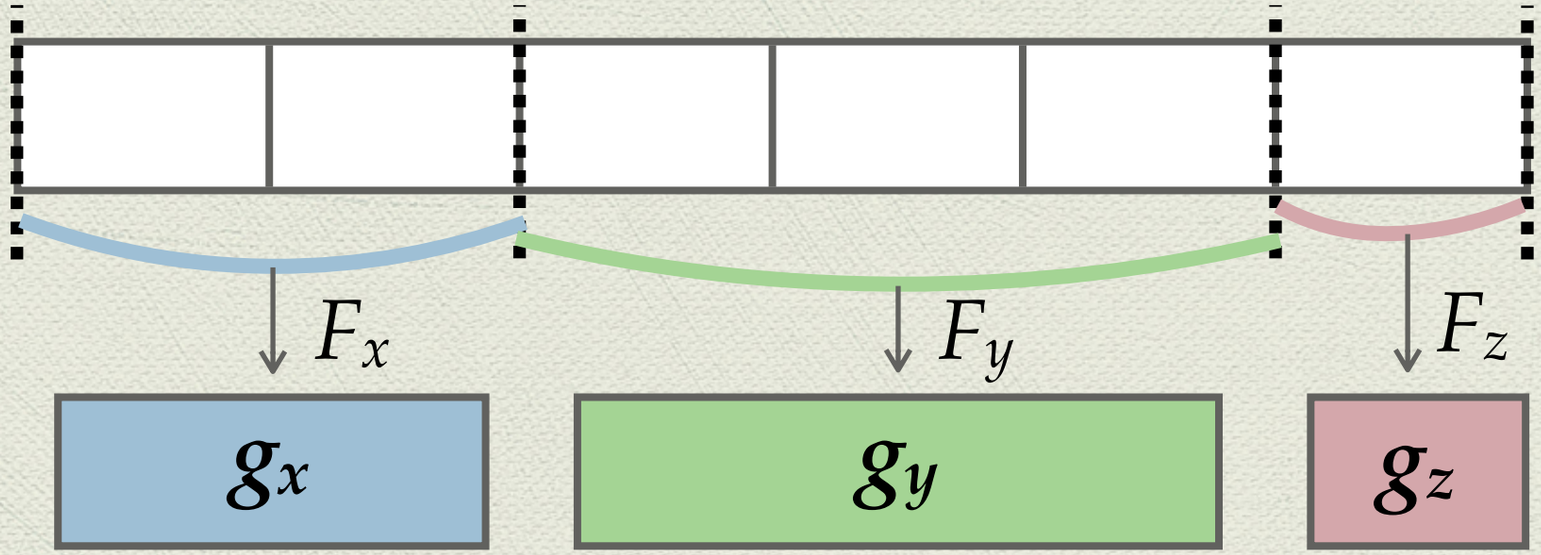
$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

$$\boldsymbol{U_i(\mathrm{p}) = \mathbf{concat}\ (g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))}$$

$$\boldsymbol{g_j(\mathrm{p}) = \mathbf{map}\ F_j\ \mathbf{subpath}(\mathrm{p}, t_j, t'_j)}$$

Input Path

Path Segments

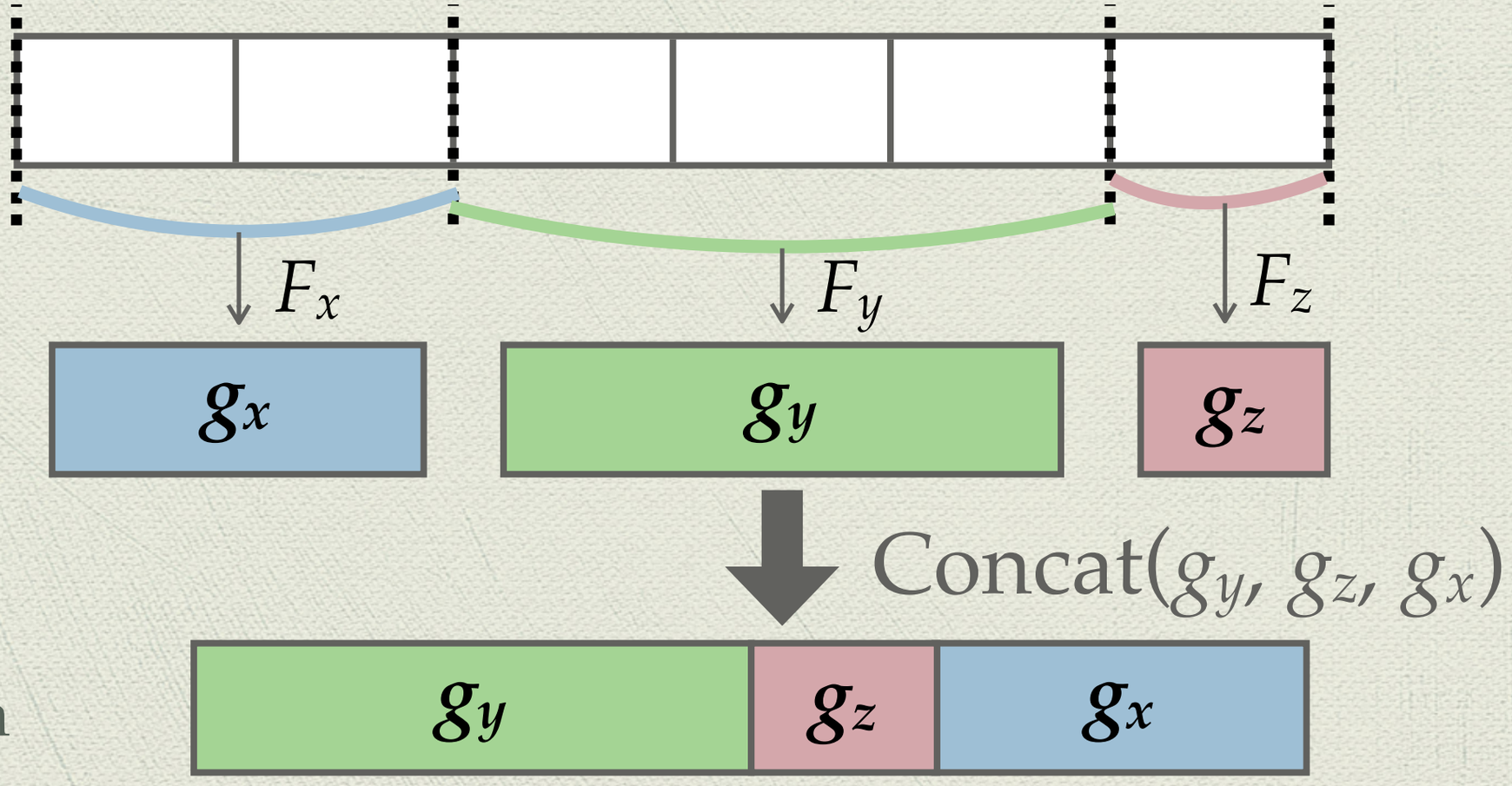


Unifier

$$\{C_1(\mathrm{p}) \rightarrow U_1(\mathrm{p}) \; ; \ldots ; \; C_n(\mathrm{p}) \rightarrow U_n(\mathrm{p})\}$$

$$\boldsymbol{U_i(\mathrm{p}) = \mathbf{concat}\,(g_1(\mathrm{p}), \ldots, g_m(\mathrm{p}))}$$

$$\boldsymbol{g_j(\mathrm{p}) = \mathbf{map}\; F_j\; \mathbf{subpath}(\mathrm{p}, t_j, t'_j)}$$



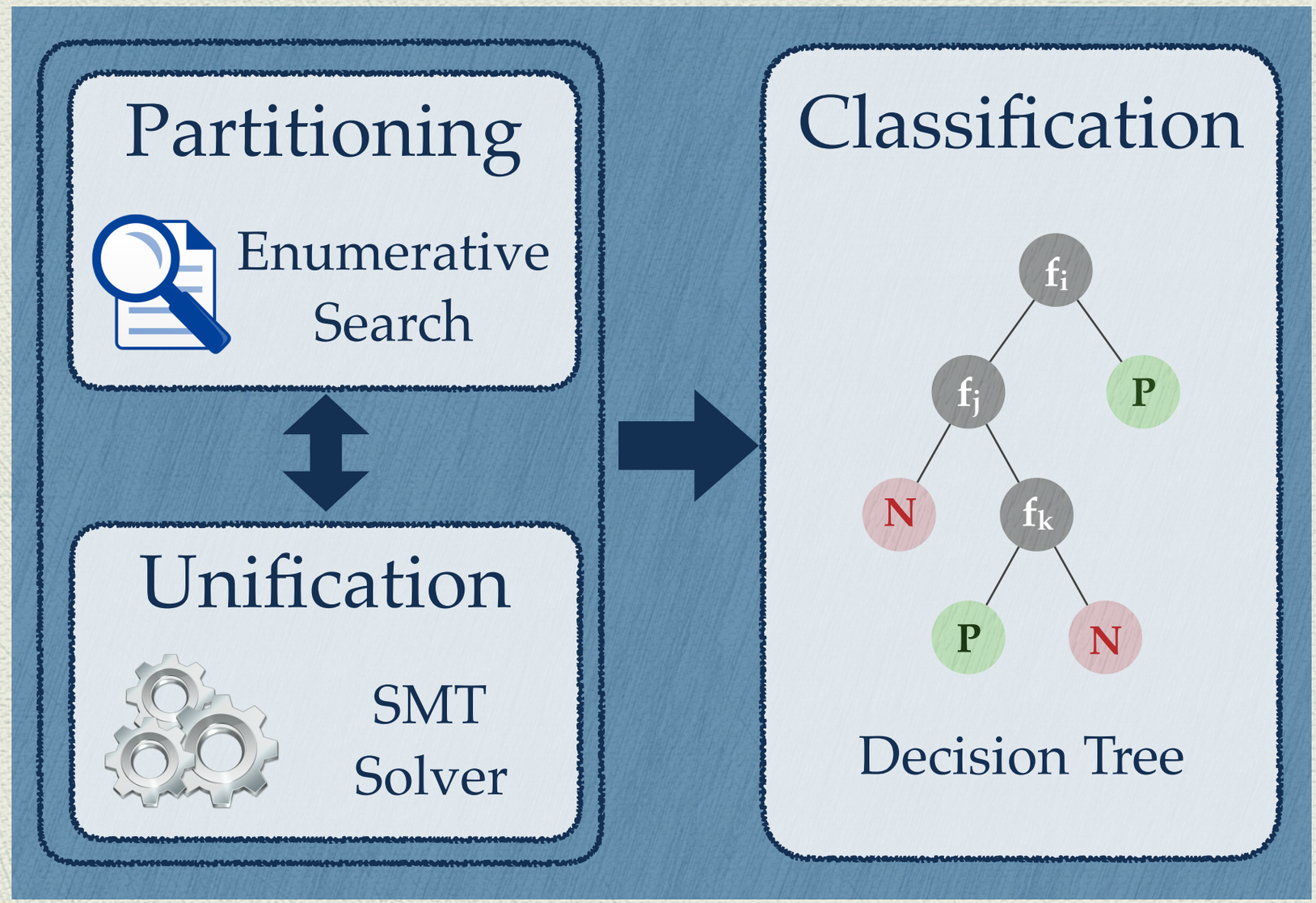
Goal of Synthesis

- Generalization: Find the simplest path transformer.
  - Simplest ≡ minimum number of conditionals

Goal of Synthesis

- Generalization: Find the simplest path transformer.
  - Simplest ≡ minimum number of conditionals
- Prevents synthesis of a transformer overfitted to examples.

Path Transformer Synthesizer

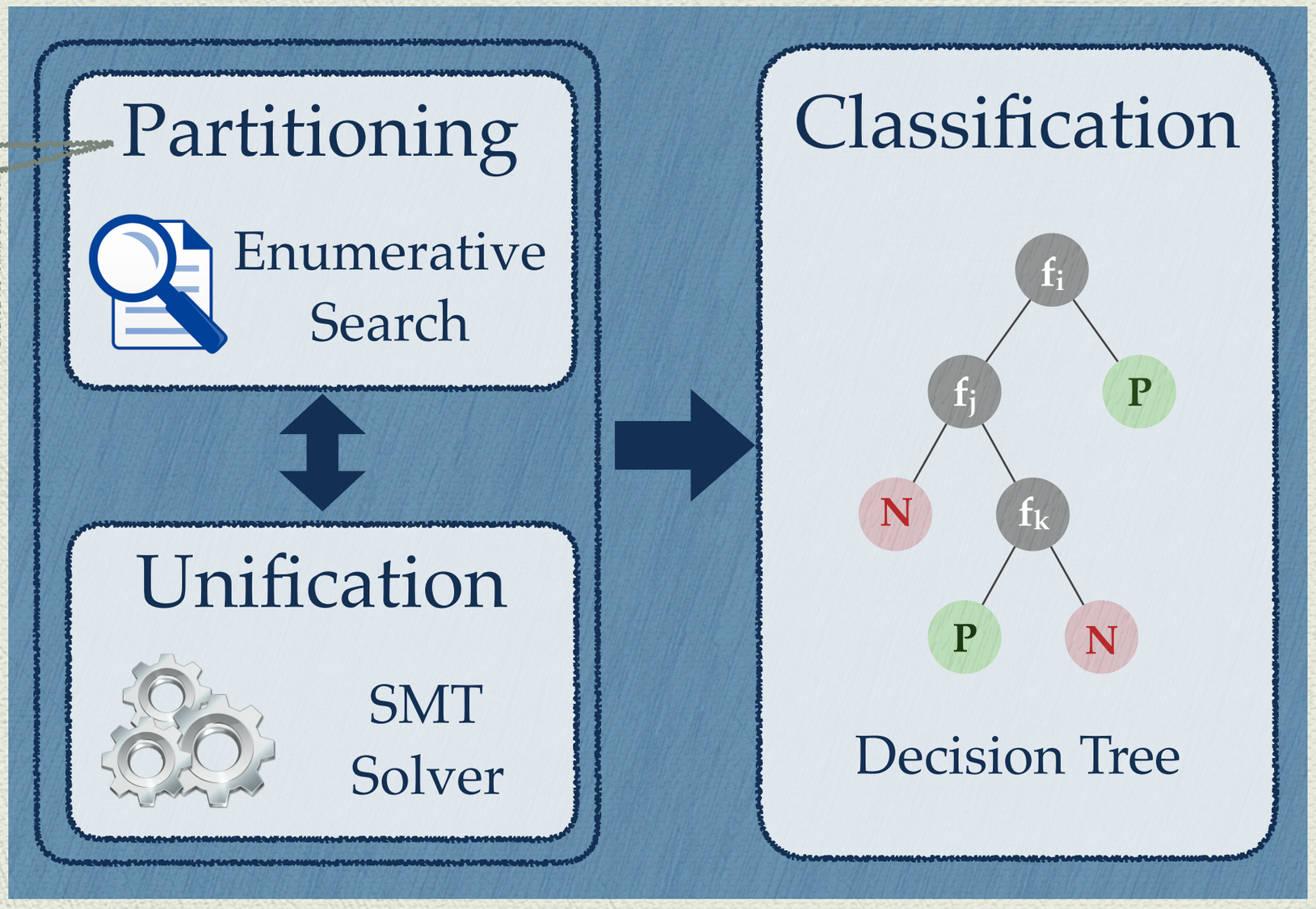


Path Transformer Language

$$\{C_1 \rightarrow U_1 ; \ldots ; C_i \rightarrow U_i ; \ldots ; C_n \rightarrow U_n \}$$

Path Transformer Synthesizer

Find the number of conditionals



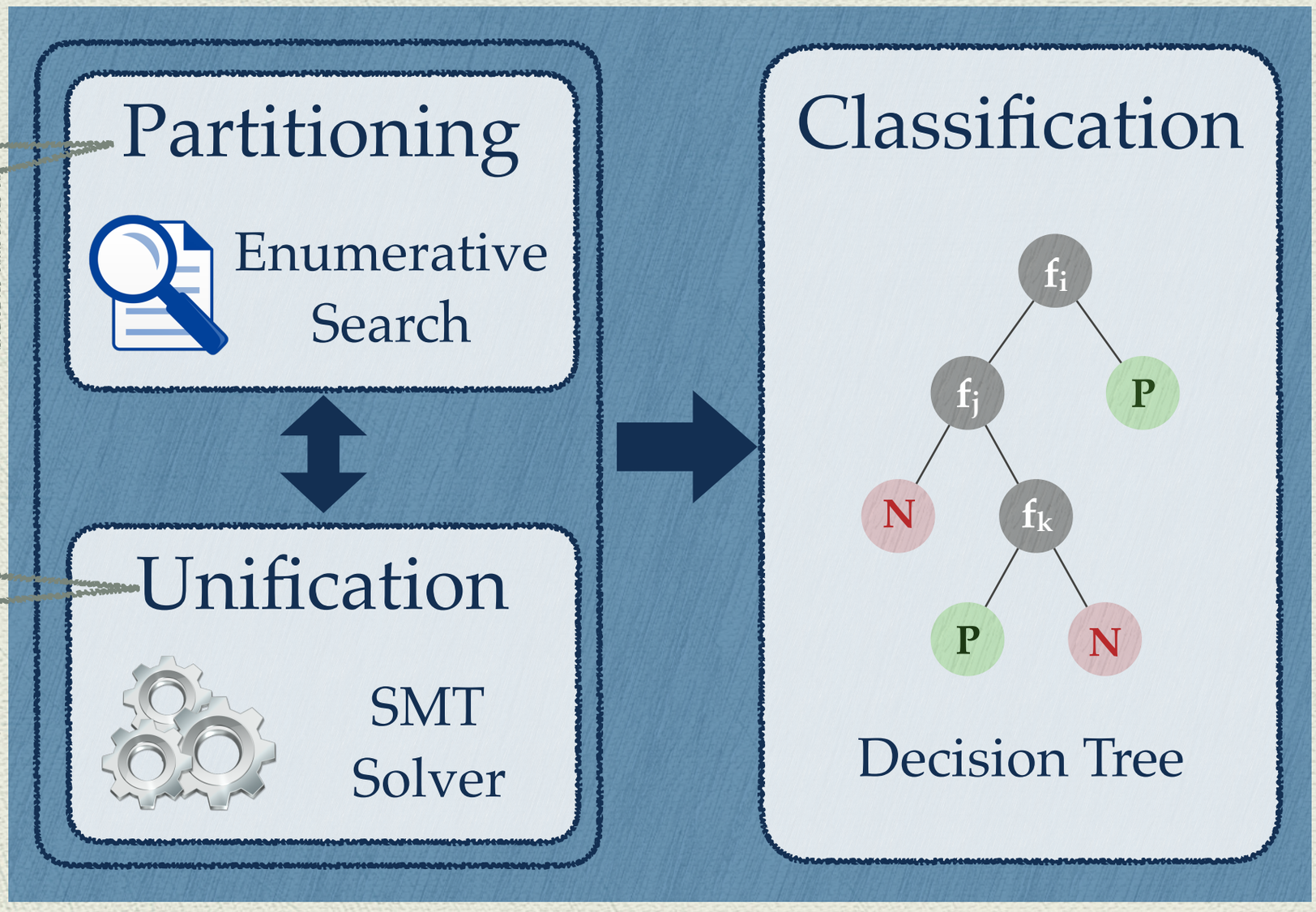
Path Transformer Language

$$\{C_1 \rightarrow U_1 ; \ldots ; C_i \rightarrow U_i ; \ldots ; C_n \rightarrow U_n \}$$

Path Transformer Synthesizer

Find the number of conditionals

Find unifier U_i for each partition



Path Transformer Language

$$\{C_1 \rightarrow U_1 ; \ldots ; C_i \rightarrow U_i ; \ldots ; C_n \rightarrow U_n \}$$

Path Transformer Synthesizer

Find the number of conditionals

Partitioning

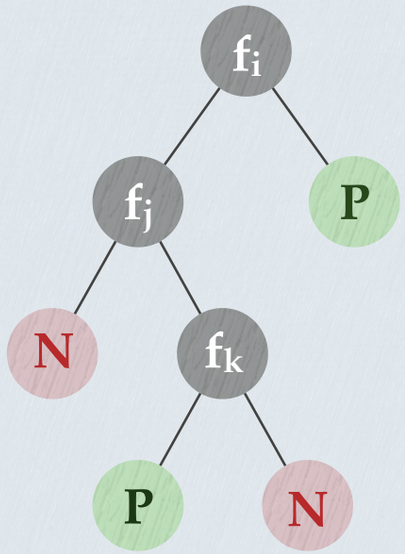
Enumerative Search

Unification

SMT Solver

Find unifier U_i for each partition

Classification



Decision Tree

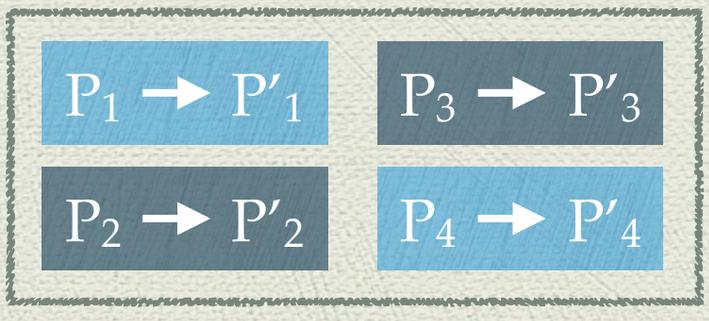
Find predicate C_i for each partition

Path Transformer Language

$$\{C_1 \rightarrow U_1 ; \ldots ; C_i \rightarrow U_i ; \ldots ; C_n \rightarrow U_n \}$$

Partitioning

Divide examples to minimum number of unifiable subsets



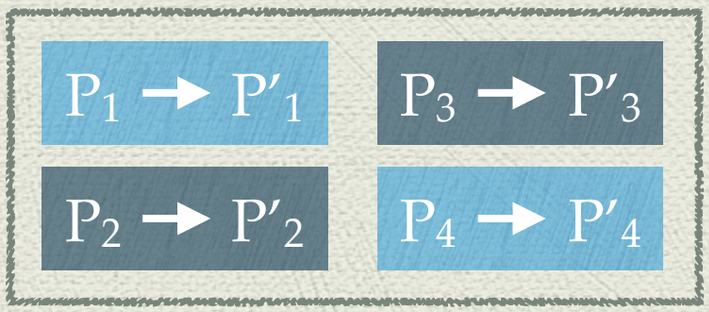
G $P'_1 = G(P_1)$ $P'_4 = G(P_4)$

F $P'_2 = F(P_2)$ $P'_3 = F(P_3)$

Partitioning

Divide examples to minimum number of unifiable subsets

- Conduct **enumerative search:**
  - Increase number of subsets until a solution is found.



G $P'_1 = G(P_1)$ $P'_4 = G(P_4)$

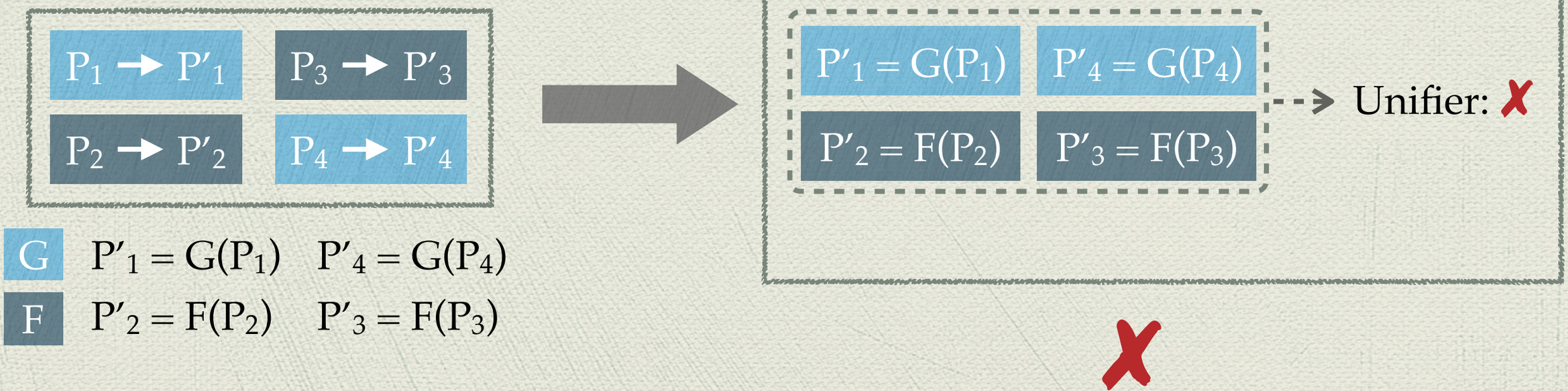
F $P'_2 = F(P_2)$ $P'_3 = F(P_3)$

Partitioning

Divide examples to <u>minimum</u> number of unifiable subsets

- Conduct **enumerative search:**
  - Increase number of subsets until a solution is found.

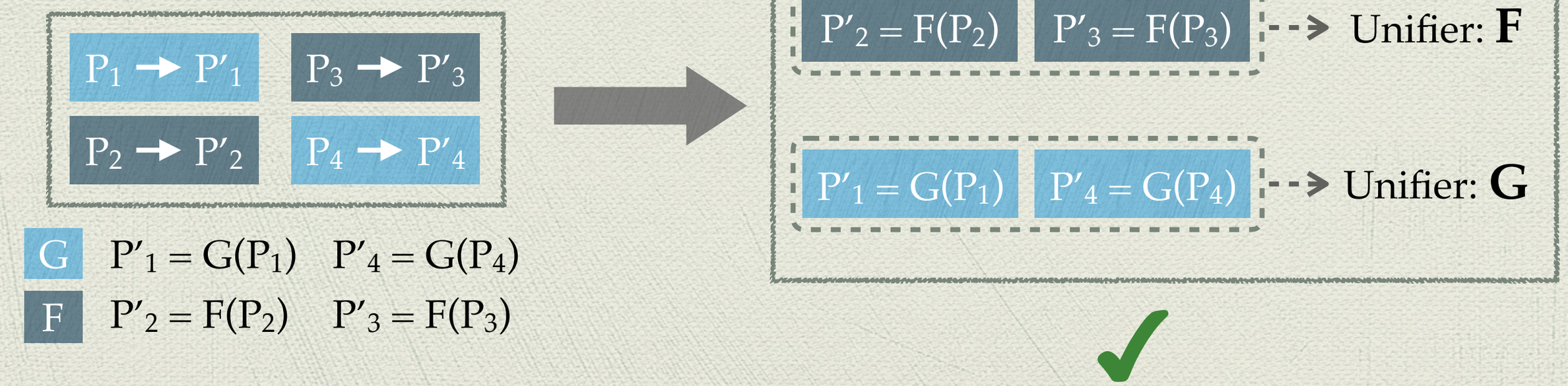
number of subsets = **1**



Partitioning

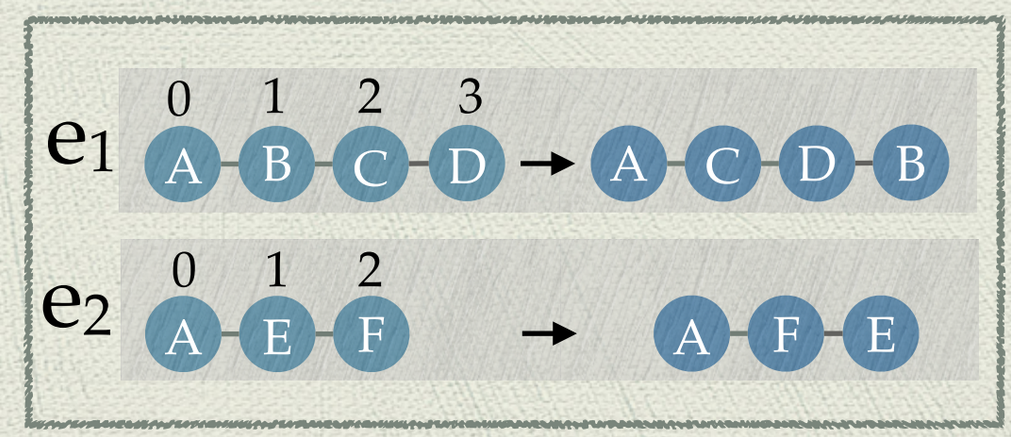
Divide examples to <u>minimum</u> number of unifiable subsets

- Conduct **enumerative search:**
  - Increase number of subsets until a solution is found.



Unification

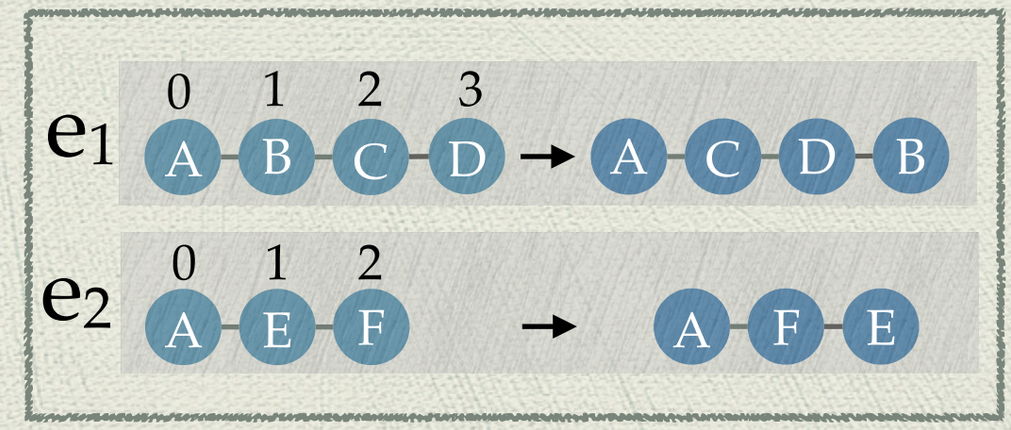
Determine if a subset of examples has a *unifier* or not



Unification

Determine if a subset of examples has a *unifier* or not

- Represent examples using numerical representation.
  - Output path represented as a permutation of input path.



e_1.out = [0, 2, 3, 1]

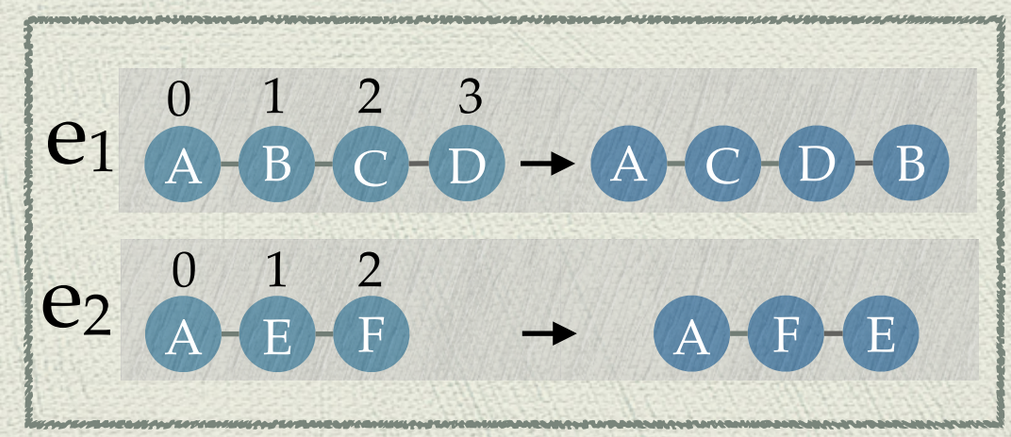
e_2.out = [0, 2, 1]

Unification

Determine if a subset of examples has a *unifier* or not

- Represent examples using numerical representation.
  - Output path represented as a permutation of input path.

- Use **SMT solver**:
  - Check if there exists a unifier for all examples.



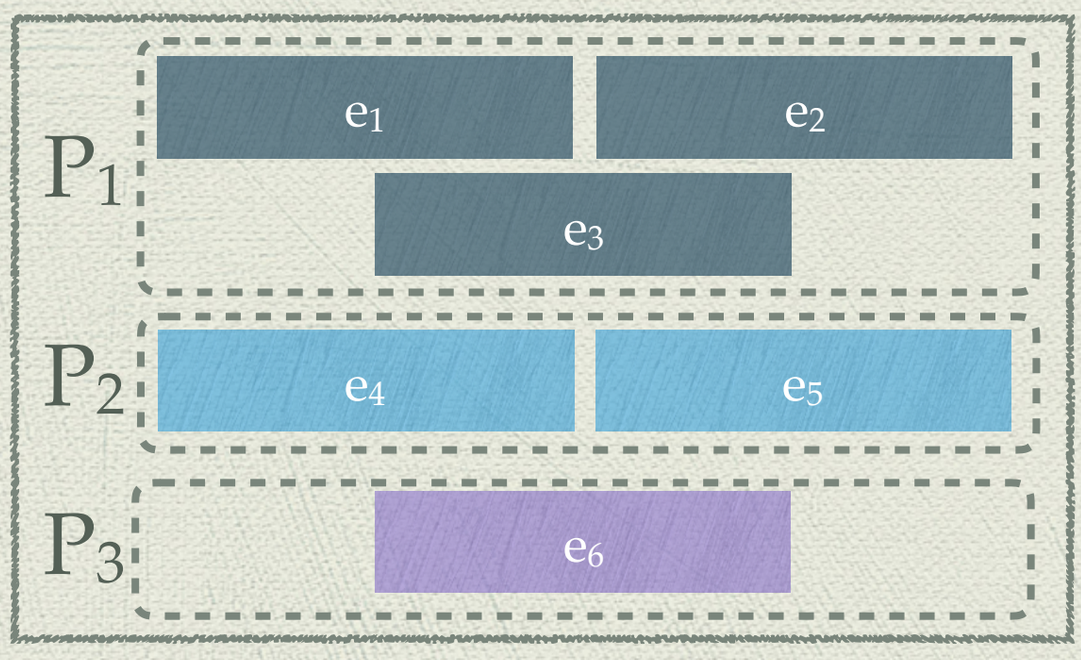
$e_1.out = [0, 2, 3, 1]$

$e_2.out = [0, 2, 1]$

Unifiable ✓

Classification

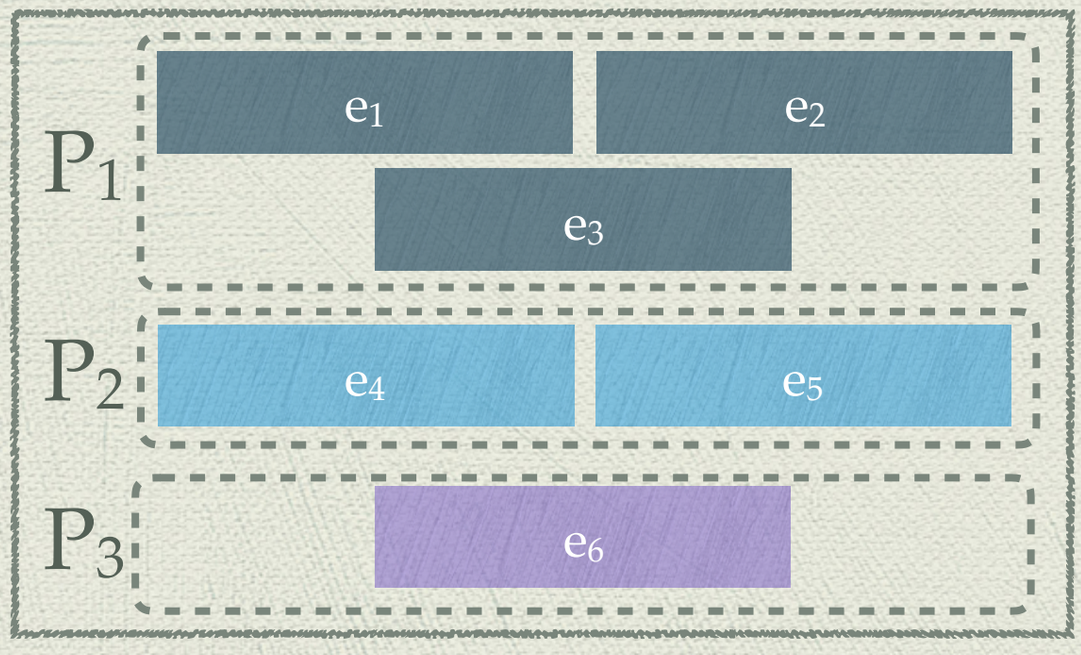
For each partition, find a predicate that differentiates examples in that partition from the rest.



Classification

For each partition, find a predicate that differentiates examples in that partition from the rest.

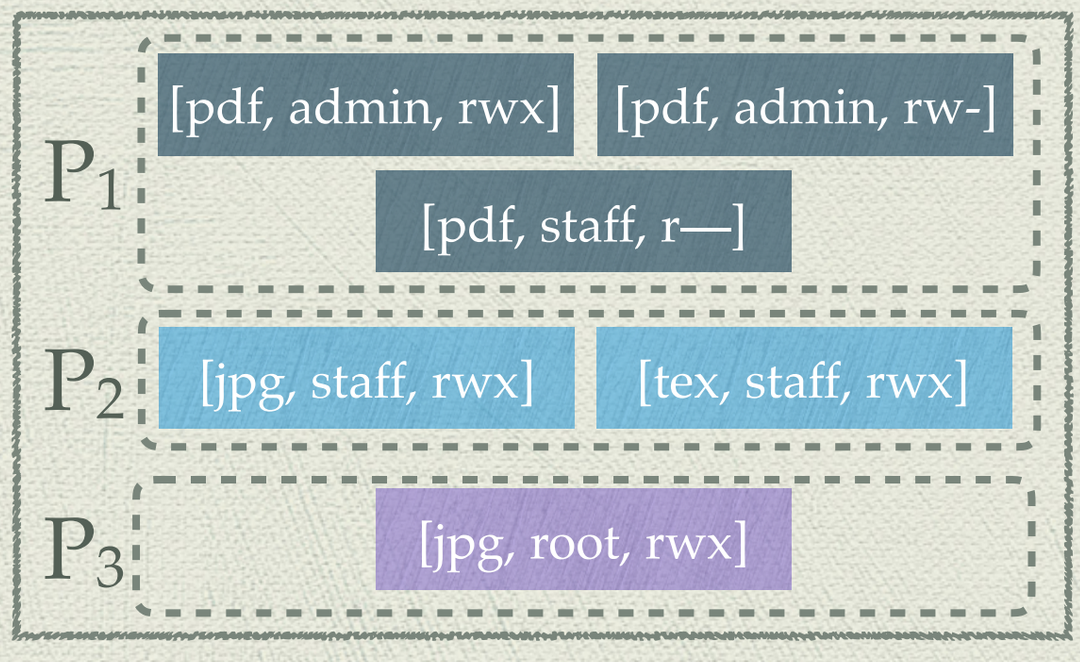
- Use **decision tree learning**: [ID3 algorithm, J. R. Quinlan]
  - Represent each example as a set of features.



Classification

For each partition, find a predicate that differentiates examples in that partition from the rest.

- Use **decision tree learning**: [ID3 algorithm, J. R. Quinlan]
  - Represent each example as a set of features.

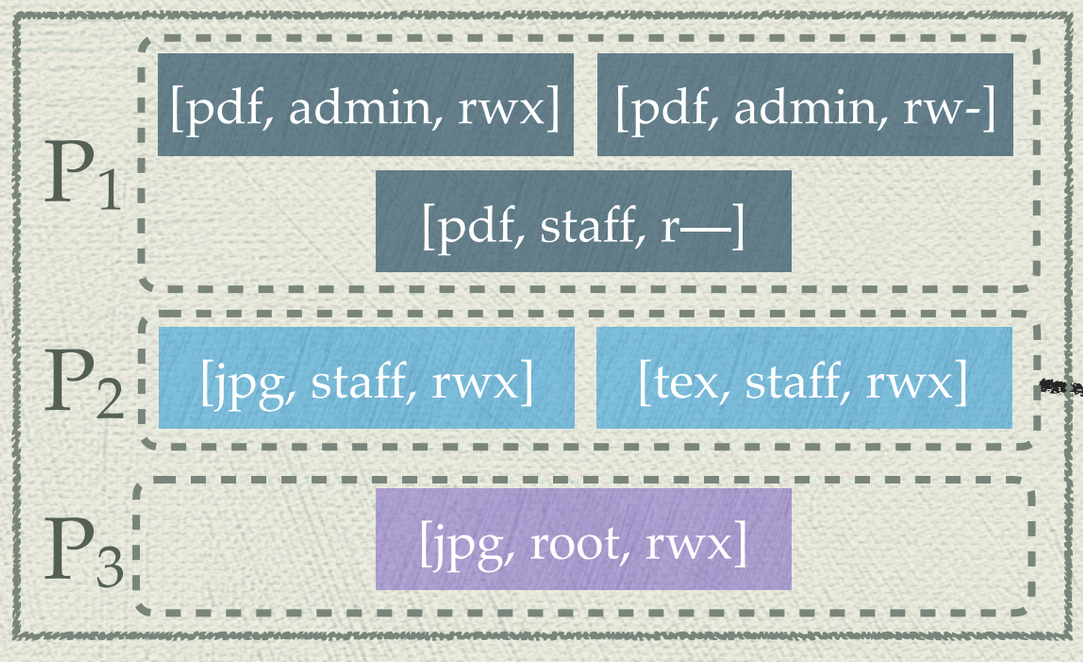


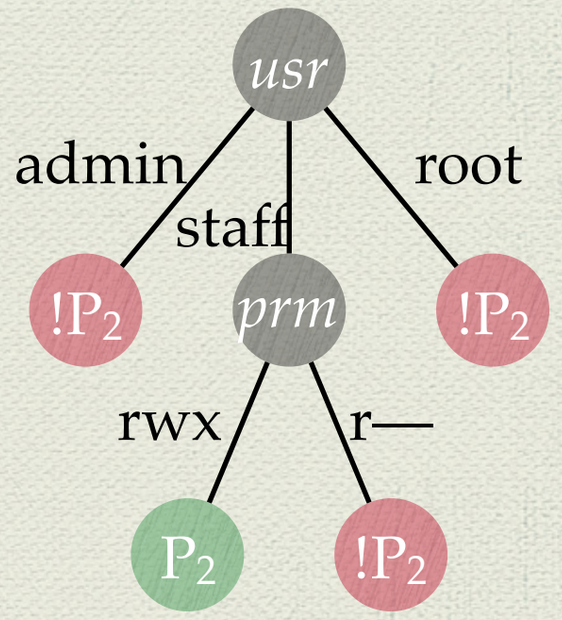
Feature set: [extension, user, permissions]

Classification

For each partition, find a predicate that differentiates examples in that partition from the rest.

- Use **decision tree learning**: [ID3 algorithm, J. R. Quinlan]
  - Represent each example as a set of features.





Feature set: [extension, user, permissions]

(*user* = staff) &&
(*permissions* = rwx)

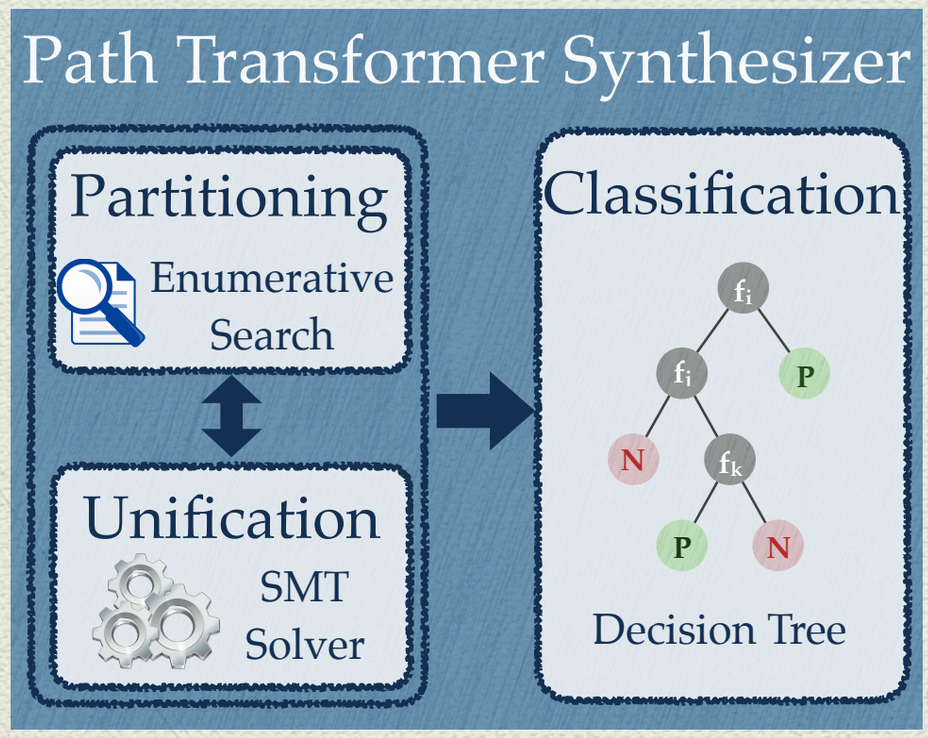
Technical Contributions

- Reduction of tree transformation synthesis to path transformation synthesis

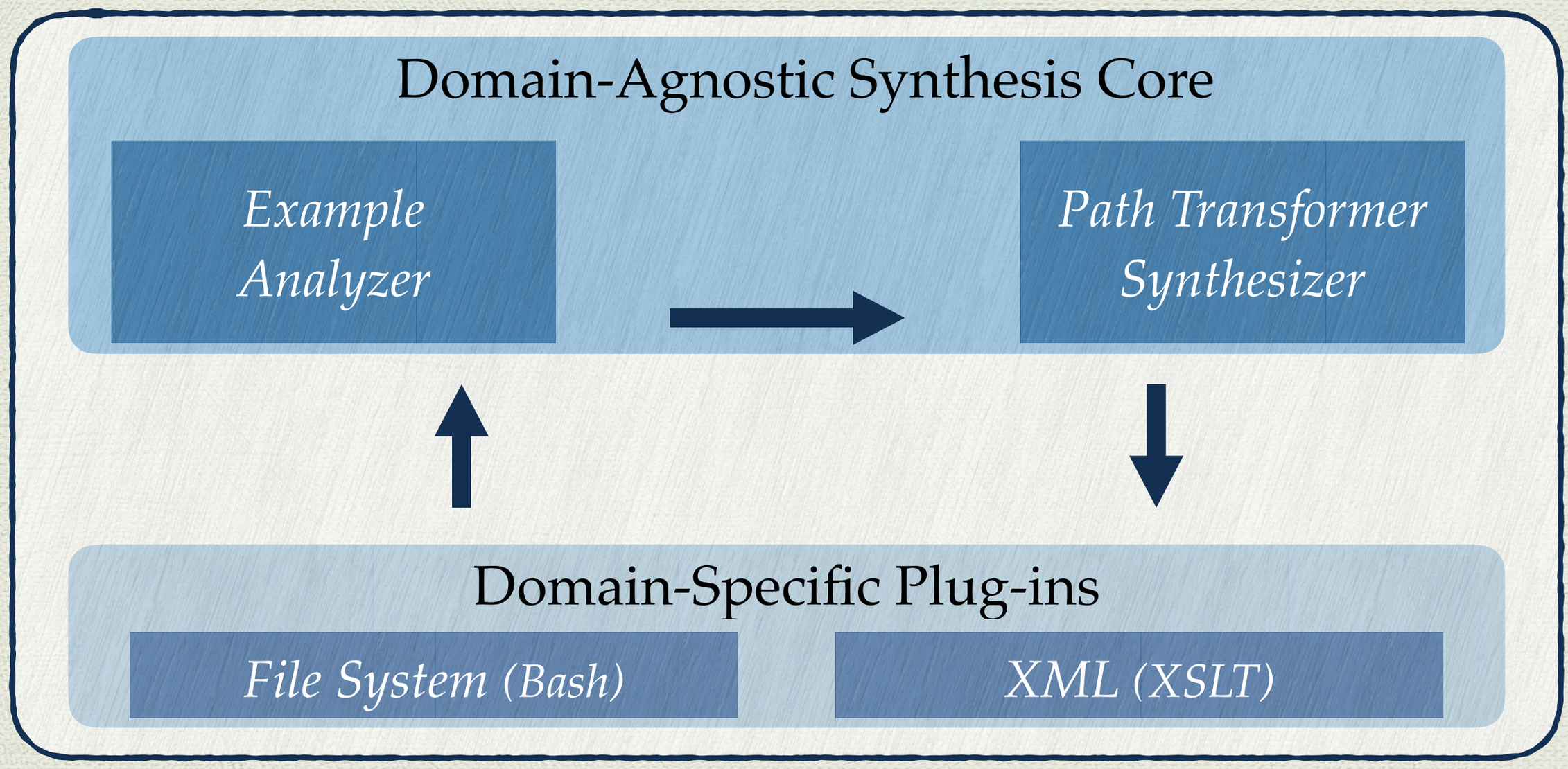
Technical Contributions

- Reduction of tree transformation synthesis to path transformation synthesis

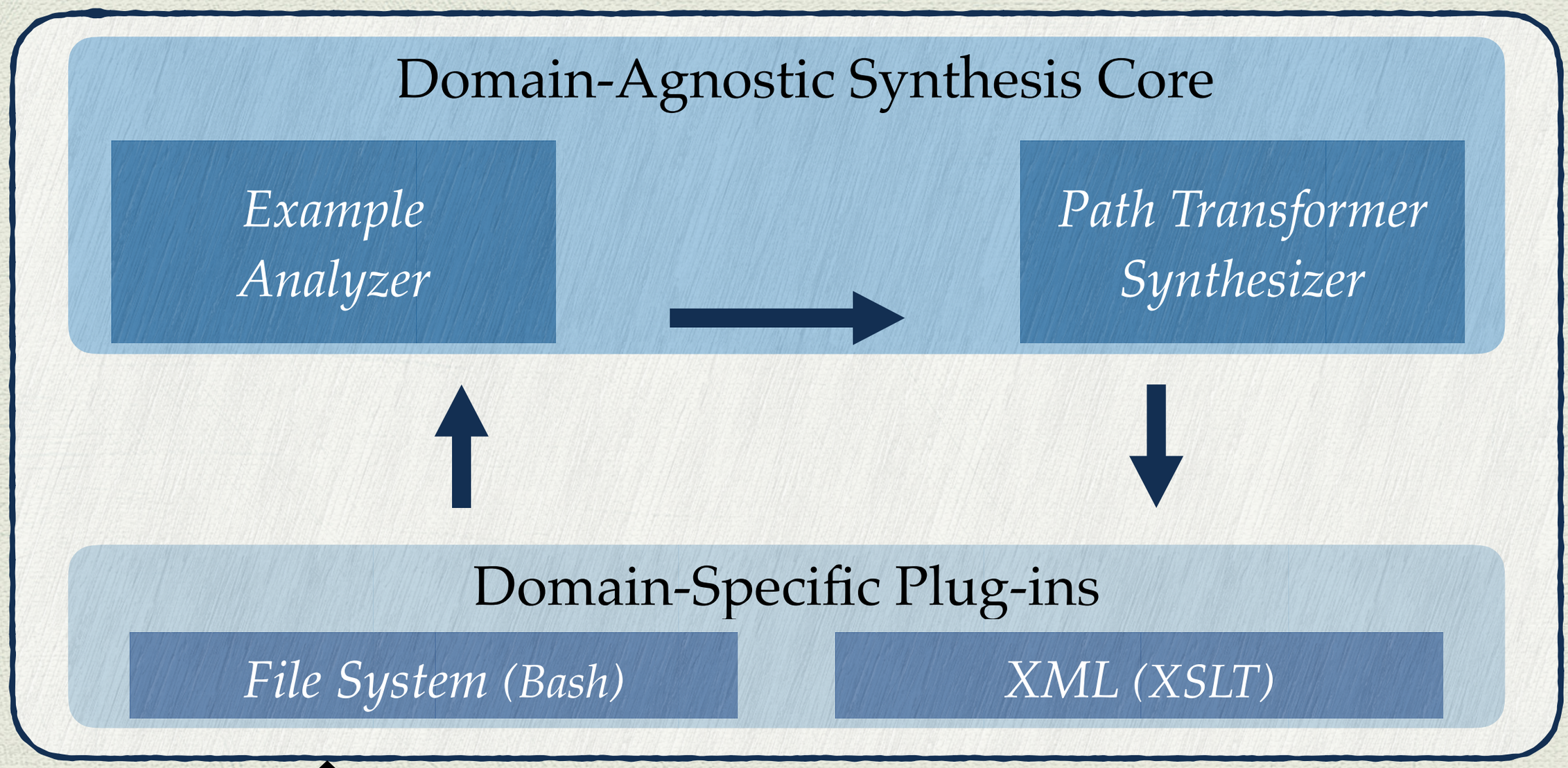
- New algorithm for path transformation synthesis based on:
  - **Enumerative Search**
  - **SMT Solving**
  - **Decision Tree Learning**

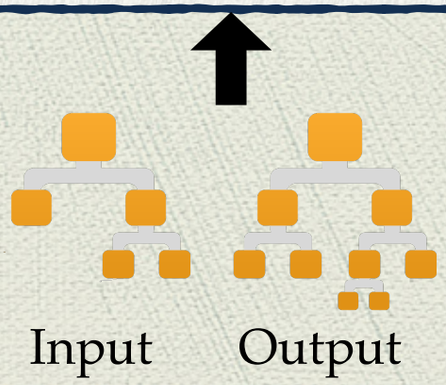


HADES Design

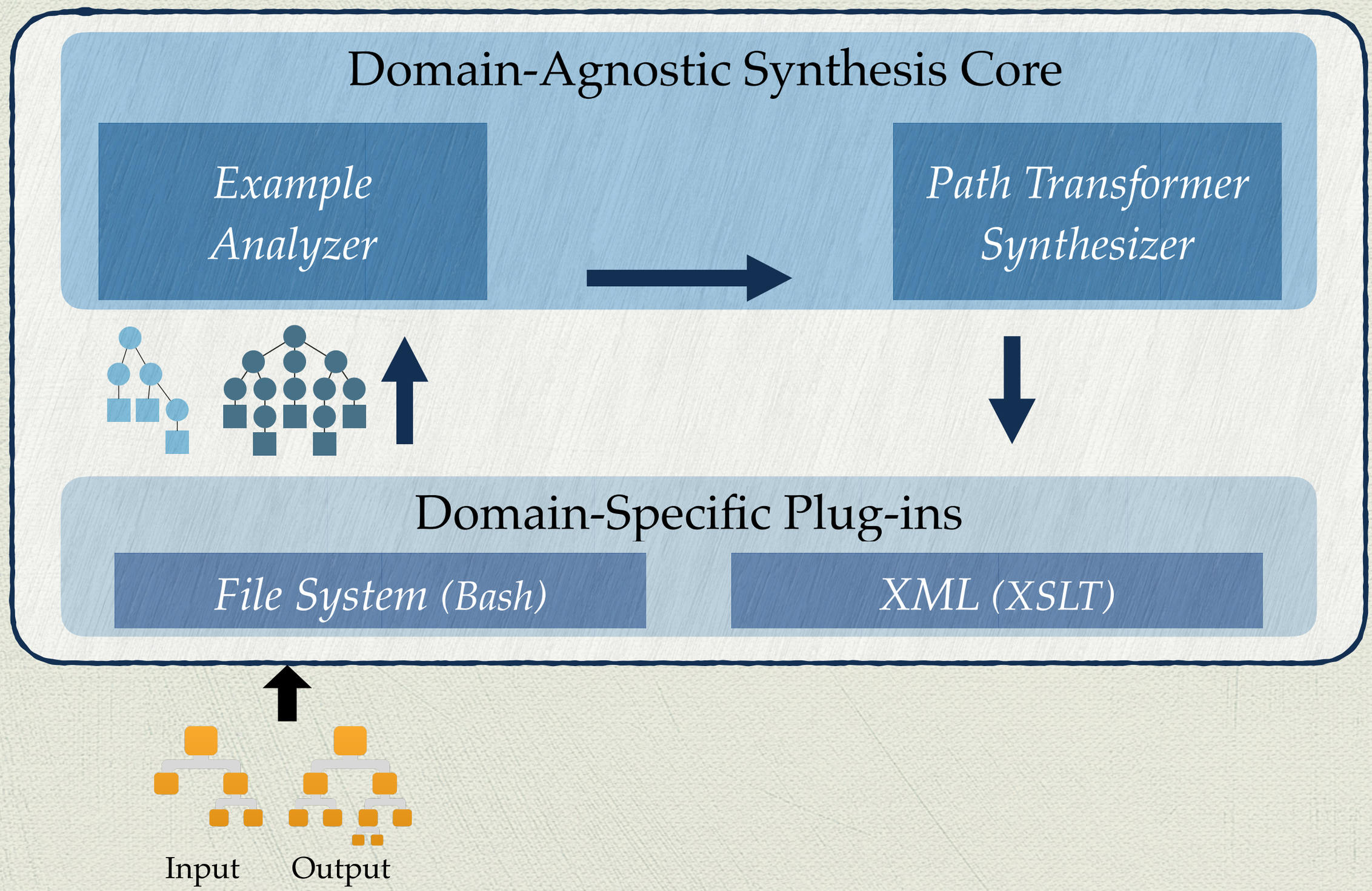


HADES Design





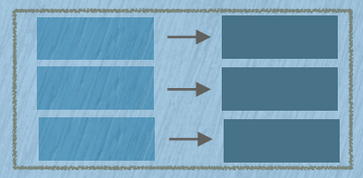
HADES Design



HADES Design

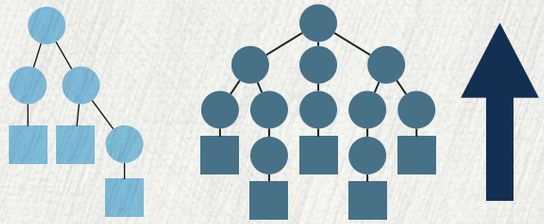
Domain-Agnostic Synthesis Core

Example Analyzer



Path Examples

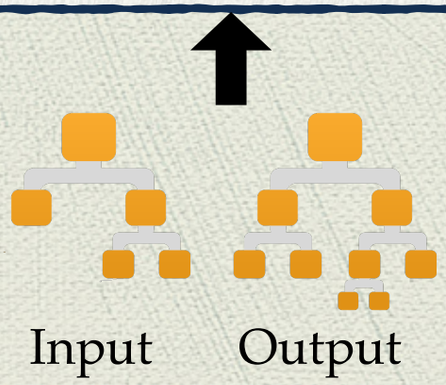
Path Transformer Synthesizer



Domain-Specific Plug-ins

File System (Bash)

XML (XSLT)

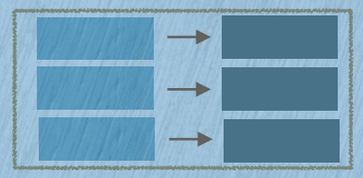


Input Output

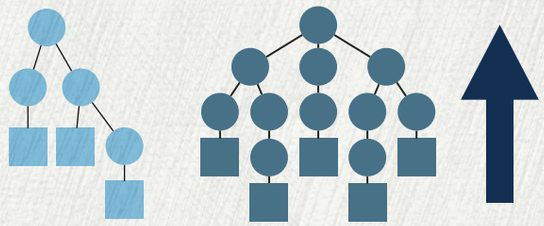
HADES Design

Domain-Agnostic Synthesis Core

Example Analyzer



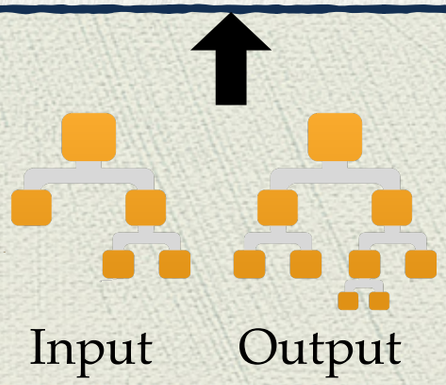
Path Transformer Synthesizer



Domain-Specific Plug-ins

File System (Bash)

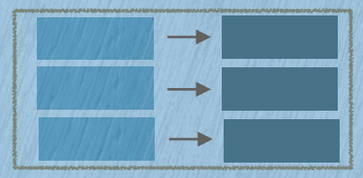
XML (XSLT)



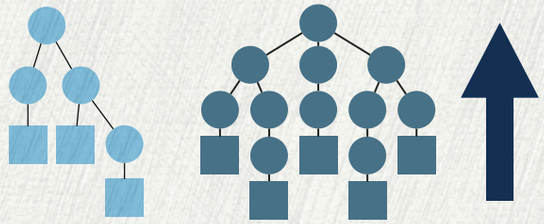
HADES Design

Domain-Agnostic Synthesis Core

Example Analyzer



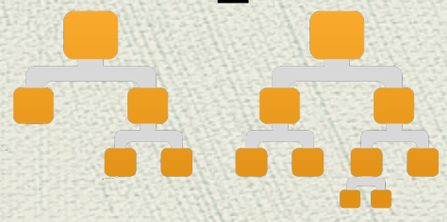
Path Transformer Synthesizer



Domain-Specific Plug-ins

File System (Bash)

XML (XSLT)



Input Output

Executable code

Demo

Evaluation Criteria

Expressiveness

Performance

Usability

Evaluation Criteria

Expressiveness

Can it synthesize transformations for real-world tasks?

Performance

Usability

Evaluation Criteria

Expressiveness

Can it synthesize transformations for real-world tasks?

Performance

How long does it take to synthesize the desired transformation?

Usability

Evaluation Criteria

Expressiveness

Can it synthesize transformations for real-world tasks?

Performance

How long does it take to synthesize the desired transformation?

Usability

How many rounds of user interaction does it need?

Evaluation Setup

- Collected **36** benchmarks: **24** *File System* and **12** *XML*
  - From online forums (e.g. stack overflow, bashscript.org)

Evaluation Setup

- Collected **36** benchmarks: **24** *File System* and **12** *XML*
  - From online forums (e.g. stack overflow, bashscript.org)

- Conducted a **user study** with **6** participants
  - Half of participants did not have any programming knowledge.

Evaluation Setup

- Collected **36** benchmarks: **24** *File System* and **12** *XML*
  - From online forums (e.g. stack overflow, bashscript.org)

- Conducted a **user study** with **6** participants
  - Half of participants did not have any programming knowledge.

- Users provide example transformations and validate result.
  - If not satisfied, repeat the process with a new set of examples.

Evaluation Setup

- Collected **36** benchmarks: **24** *File System* and **12** *XML*
  - From online forums (e.g. stack overflow, bashscript.org)

- Conducted a **user study** with **6** participants
  - Half of participants did not have any programming knowledge.

- Users provide example transformations and validate result.
  - If not satisfied, repeat the process with a new set of examples.

- All experiments conducted on a commercial MacBook.

Results

Expressiveness

Synthesized the desired transformation for **all benchmarks**

Performance

Usability

Results

Expressiveness

Synthesized the desired transformation for **all benchmarks**

Performance

Less than **one second** for **91.6%** of benchmarks
(longest took **18s**)

Usability

Results

Expressiveness

Synthesized the desired transformation for **all benchmarks**

Performance

Less than **one second** for **91.6%** of benchmarks

(longest took **18s**)

Usability

88.8% of benchmarks require only **1 or 2 rounds** of user interaction

Example complexity: 1 input-output tree with 4 leaves (avg)

Conclusion

- A general approach for tree transformation synthesis.
- Can synthesize a broad class of transformations which alter the tree structure.
- Promising results on real-world synthesis tasks.