Synthesizing Transformations on Hierarchically Structured Data *

Navid Yaghmazadeh
University of Texas at Austin
nyaghma@cs.utexas.edu

Christian Klinger †
University of Freiburg
klingerc@informatik.uni-freiburg.de

Isil Dillig
University of Texas at Austin
isil@cs.utexas.edu

Swarat Chaudhuri
Rice University
swarat@rice.edu

Abstract

This paper presents a new approach for synthesizing transformations on tree-structured data, such as Unix directories and XML documents. We consider a general abstraction for such data, called *hierarchical data trees (HDTs)* and present a novel example-driven synthesis algorithm for HDT transformations. Our central insight is to reduce the problem of synthesizing tree transformers to the synthesis of list transformations that are applied to the paths of the tree. The synthesis problem over lists is solved using a new algorithm that combines SMT solving and decision tree learning. We have implemented our technique in a system called HADES and show that HADES can automatically synthesize a variety of interesting transformations collected from online forums.

1. Introduction

Much of the data that users deal with today are inherently hierarchical or tree-shaped. Examples include:

- *File systems:* A file system is naturally seen as a tree where directories represent internal nodes and files correspond to leaves.
- *XML documents:* In XML documents, data is organized as a tree structure, where each subtree is identified by a pair of start and end tags.
- *HDF files:* Many scientific documents are stored as HDF files that have a tree structure. In this format, *groups* correspond to internal nodes and *datasets* represent leaves.

End-users of such hierarchical data must often perform various kinds of *tree transformations* on their data. For example, a user may want to reorganize the file hierarchy in a directory tree, or change the structure of tags in an XML document.

In principle, one may accomplish these tasks by writing a program, such as a Bash or XSLT script. However, given that end users often do not have the expertise to write programs, an attractive alternative is to automatically synthesize such a program from a high-level specification. In particular, synthesis of programs from examples [12, 23] seems like a natural fit for this setting.

Motivated by this context, this paper proposes a new algorithm, and its implementation in a system called HADES, for automatically synthesizing hierarchical data transformations from input-output examples. Our algorithm operates on a general abstraction of hierarchical data, called *hierarchical data trees (HDTs)*, and does not place any restrictions on the depth or fanout of the hierarchy. Our method is able to synthesize a rich class of tree transformations that commonly arise in real-world data manipulation tasks, such as restructuring of the data hierarchy and modification of metadata.

Synthesizing programs over unbounded trees is a difficult problem. In spite of recent attempts [2, 10, 18], the problem lacks a comprehensive solution. For example, so far as we know, no existing technique can synthesize nontrivial alterations to the structure of an input tree.

Our approach to tree transformation synthesis is based on a simple but novel insight: We note that, under natural assumptions, tree transformations can be written as a *composition of transformations over the paths of the tree*. Our algorithm uses this observation to generate code that behaves as follows: (1) Given an input tree T, generate the set of paths in T; (2) apply a list transformation to each path; and (3) combine the transformed paths into a transformed tree.

An alternative to the above strategy is to generate a transformation that operates directly on the tree. While this com-

* This work was supported in part by NSF Award #1453386 and AFRL Awards # 8750-14-2-0270.

† This work was done while the author was at the University of Texas at Austin.

DOI: http://dx.doi.org/10.1145/(to come)

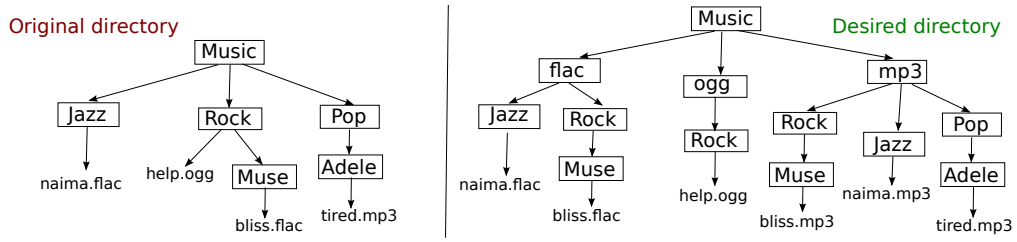


Figure 1. Input-output directories for motivating example

peting approach could conceivably generate programs that are more compact than those we produce, our approach provides a practical way to synthesize complex programs that change the *structure* of the input tree. Put another way, our approach trades off the complexity of the synthesized programs for faster and more comprehensive synthesis.

Another novelty of our approach is a new algorithm for synthesizing list transformations that combines SMT solving and decision tree learning. Given a set of input-output lists, our algorithm partitions the examples into unifiable groups, where a *unifier* is a conditional-free program containing loops. Our algorithm learns unifiers using SMT solving and uses decision tree learning to find predicates that differentiate one unifiable subset of examples from the others.

We have implemented our technique in a system called HADES, which provides a language-agnostic backend for synthesizing HDT transformations. In principle, HADES can be used to generate code in any DSL, provided it has been plugged into our infrastructure. Our current implementation provides two DSL front-ends, one for bash scripts (Unix directories), and another for XSLT (XML transformations).

We summarize our contributions as follows:

- We present a new method for synthesizing tree transformations that can change the structure of the input tree. Our key insight is to formulate tree transformations as a composition of list transformations.
- We present a new algorithm —based on SMT solving and decision tree learning— for learning list transformations.
- We implement our method in a tool called HADES and use it to synthesize bash scripts and XSLT programs for various tasks obtained from on-line forums. Our evaluation shows that HADES is practical, both in terms of running time as well as the input required from end-users.

2. Overview

We illustrate our approach using a motivating example from the file system domain. Consider a user, Bob, who has a large collection of music files organized by genres: The top-level `Music` directory has subfolders for each genre, and each subdirectory contains a collection of music files and subfolders (e.g., one for each band). Furthermore, Bob's music files come in three different formats: mp3, ogg, and flac. However, since not every music player supports all formats, Bob wants to categorize his music based on file type while also maintaining the original organization based on genres. In addition, since few applications can play music files in flac format, Bob wants to convert all his flac files to mp3 and keep both the original as well as the converted files.

Let us consider how Bob can use HADES for synthesizing a bash script that performs his desired transformation. To use HADES, Bob first constructs the input-output example shown in Figure 1. Observe that the `Music` directory in the output has three subfolders called `flac`, `ogg`, and `mp3`. Also observe that the `naima.flac` file under the `Jazz` subfolder in the input has been duplicated as `naima.flac` and `naima.mp3` in the output under the `flac/Jazz` and `mp3/Jazz` directories respectively.

Given this input, HADES first converts each of the input and output directories to an intermediate representation called a *hierarchical data tree* (HDT) and then generates a set of list transformation examples $\mathcal{E}$, as shown in Figure 2. Each example $e \in \mathcal{E}$ consists of a pair of lists (p_1, p_2) where p_1 is a root-to-leaf path in the input HDT and p_2 is a corresponding path in the output HDT. We represent paths as a list of pairs (l, d) where l is a node label and d is the data stored at that node. In this application domain, labels correspond to directory/file names, and data includes information about permissions, owner, file type etc.

After constructing the list transformation examples $\mathcal{E}$, HADES synthesizes a *path transformation function* f such that $p' \in f(p)$ for every example $(p, p') \in \mathcal{E}$. Note that we allow path transformers to return a *set* of paths in order to support duplication and deletion.

In the HADES system, the synthesis of path transformers consists of two phases: In the first phase, we *partition* the input examples into sets of *unifiable* groups, using SMT solving for unification. In the second phase, we perform *classification* by using decision tree learning to find a predicate that differentiates one unifiable group from the others. Going back to our example, HADES partitions examples $\mathcal{E}$ into two groups $\mathcal{P}_1 = \{\mathcal{E}_1, \mathcal{E}_3, \mathcal{E}_4, \mathcal{E}_6\}$ and $\mathcal{P}_2 = \{\mathcal{E}_2, \mathcal{E}_5\}$ and infers the following unifier χ_1 for partition $\mathcal{P}_1$:

$$\begin{array}{ll} \text{concat (} & \text{map Id subpath}(x, 1, 1), \\ & \text{map ExtOf subpath}(x, \text{size}(x), \text{size}(x)), \\ & \text{map Id subpath}(x, 2, \text{size}(x)) \quad \text{)} \end{array}$$

Here, $\text{subpath}(x, t, t')$ yields a subpath of x between indices t and t', Id is the identity function, and ExtOf yields the extension (i.e., file type) for a given node. Similarly,

$$
\begin{array}{lll}
\mathcal{E}_1: & p_1 = [(\text{Music}, d_m), (\text{Jazz}, d_j), (\text{naima}, d_n)] & \mapsto \quad p_1' = [(\text{Music}, d_m), (\text{flac}, d_f), (\text{Jazz}, d_j), (\text{naima}, d_n)] \\
\mathcal{E}_2: & p_1 = [(\text{Music}, d_m), (\text{Jazz}, d_j), (\text{naima}, d_n)] & \mapsto \quad p_1'' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Jazz}, d_j), (\text{naima}, d_n')] \\
\mathcal{E}_3: & p_2 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{help}, d_h)] & \mapsto \quad p_2' = [(\text{Music}, d_m), (\text{ogg}, d_o), (\text{Rock}, d_r), (\text{help}, d_h)] \\
\mathcal{E}_4: & p_3 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b)] & \mapsto \quad p_3' = [(\text{Music}, d_m), (\text{flac}, d_f), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b)] \\
\mathcal{E}_5: & p_3 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b)] & \mapsto \quad p_3'' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_b')] \\
\mathcal{E}_6: & p_4 = [(\text{Music}, d_m), (\text{Pop}, d_p), (\text{Adele}, d_a), (\text{tired}, d_t)] & \mapsto \quad p_4' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Pop}, d_p), (\text{Adele}, d_a), (\text{tired}, d_t)]
\end{array}
$$

Figure 2. Path transformation examples constructed by HADES

```
srcDir=$1
for inputFile in \$srcDir/*  do
  elems=$(split $inputFile)
  size=$(SizeOf $elems)
  output=concat($inputElems[0],$(Ext $elems[$size-1]),
                subList(1, $size-1, $elems))
  outputPaths+=\$output
  if {[[ $(Ext $inputFile) == flac]]} then
    output=concat($elems[0], "mp3",
           subList(1, $size-2, $elems),
           $(convertFormat $elems[$size-1]))
    outputPaths+=$output
  fi
done
makeDirectories $outputPaths
```

Figure 3. Bash script synthesized by HADES

for partition $\mathcal{P}_2$, we infer the following unifier χ_2, where FlacToMp3 is a function for converting flac files to mp3:

$$
\begin{array}{ll}
\text{concat}\,(& \text{map Id subpath}(x, 1, 1),\ \text{“mp3”}, \\
& \text{map Id subpath}(x, 2, \text{size}(x) - 1), \\
& \text{map FlacToMp3 subpath}(x, \text{size}(x), \text{size}(x))\quad)
\end{array}
$$

Next, HADES performs classification to infer a predicate characterizing the input paths in each partition. Since the input paths in partition $\mathcal{P}_1$ include all paths in the input tree, HADES infers the classifier $\phi_1 : \mathit{true}$ for $\mathcal{P}_1$. On the other hand, since partition $\mathcal{P}_2$ only includes p_1, p_3, HADES infers $\phi_2 : \text{ext} = \text{“flac”}$ as a classifier for $\mathcal{P}_2$. Hence, the overall path transformer inferred by our method is $\pi : \lambda x.\ (\chi_1;\ \text{if}(\text{ext} = \text{“flac”})\ \text{then}\ \chi_2)$.

As a final step, HADES uses this list transformer π to synthesize the tree transformation shown as (pseduo-) bash code in Figure 3. In essence, the synthesized program constructs the output directory by applying transformation π to every path in the input directory. Going back to our motivating scenario, Bob can now apply this bash script to his very large music collection and obtain the desired transformation.

3. Hierarchical Data Trees

In this section, we introduce *hierarchical data trees (HDT)* which our system uses as the canonical representation for various kinds of hierarchical data.

Definition 1. (HDT) Assume a universe Id of *labels* for tree nodes and a universe Dat of *data*. A *hierarchical data tree* T is a rooted tree represented as a quadruple (V, E, L, D) where V is a set of nodes, and E is a set of directed edges. The *labeling function* $L : V \to \text{Id}$ assigns a label to each node $v \in V$, and the *data store* $D : V \to \text{Dat}$ maps each node $v \in V$ to the data associated with v.

We emphasize that the labeling function L does not need to be one-to-one. That is, it is possible that $L(v) = L(v')$ for two distinct nodes v, v'. We write $L(V)$ to indicate the multi-set $\{\ell \mid v \in V \wedge L(v) = \ell\}$ and $L(E)$ to denote the multi-set $\{(\ell, \ell') \mid (v, v') \in E \ \wedge\ L(v) = \ell \wedge L(v') = \ell'\}$.

Example 1. File system directories can be viewed as HDTs where vertices are files and directories, and an edge from v_1 to v_2 means that v_1 is v_2's parent directory. The label for each node v is the name of the file or directory associated with v. The data store D assigns each node to its corresponding meta-data (e.g., permissions, creation date etc.).

Example 2. We can view XML files as HDTs where nodes correspond to XML elements. An edge from v to v' means that v' is nested directly inside element v. The labeling function L maps each element v to a label (s, i) where s is the name of the tag associated with v and i indicates that v is the i'th element with tag s under v's parent. The data store D maps each element v to its attributes.

Definition 2. (Well-formedness) We say that an HDT is *well-formed* iff no two sibling vertices have the same label.

Throughout this paper, we assume that HDTs are well-formed and use the term “tree” to mean a well-formed HDT. This well-formedness assumption is a lightweight restriction that applies to many real-world domains. For example, file system directories satisfy the well-formedness assumption because there cannot be two files or directories with the same name under the same directory. XML documents also satisfy this assumption because the order in which tags appear in a document is significant; hence, we can assign two different labels to sibling elements with the same tag name (recall the labeling function from Example 2).

Definition 3. (Path) A *path* p in an HDT $T = (V, E, L, D)$ is a list $[(\ell_1, d_1), \ldots, (\ell_k, d_k)]$ such that:

- $\ell_1 = L(r)$ and $d_1 = D(r)$, where r is the root of T
- $\ell_k = L(v)$ and $d_k = D(v)$, where v is a leaf of T
- For each $i \in [1, k)$, there is an edge $(v, v') \in E$ where $L(v) = \ell_i$, $L(v') = \ell_{i+1}$, $D(v) = d_i$, and $D(v') = d_{i+1}$.

Given a path $p = [(\ell_1, d_1), \ldots, (\ell_k, d_k)]$, we write $p[i].\ell$ and $p[i].d$ to indicate ℓ_i and d_i respectively. The set of paths

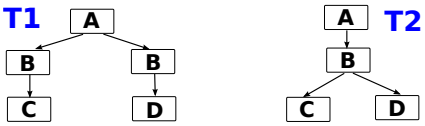


Figure 4. Example to motivate well-formedness

in T is denoted by *paths*(T), and we write *pathTo*(T, v) to denote a path starting at T's root and ending in v.

Definition 4. (Equivalence) Let $T_1 = (V_1, E_1, L_1, D_1)$ with root v_1 and $T_2 = (V_2, E_2, L_2, D_2)$ with root node v_2. We say that T_1 is *equivalent to* T_2, written $T_1 \equiv T_2$, iff the following conditions hold:

1. $L_1(v_1) = L_2(v_2)$ and $D_1(v_1) = D_2(v_2)$
2. $L_1(\text{children}(v_1)) = L_2(\text{children}(v_2))$
3. For every $(v_1', v_2') \in \text{children}(v_1) \times \text{children}(v_2)$ such that $L_1(v_1') = L_2(v_2'), \text{subtree}(T_1, v_1') \equiv \text{subtree}(T_2, v_2')$.

Intuitively, two HDTs T_1 and T_2 are equivalent if they are indistinguishable with respect to the labeling functions (L_1, L_2) and data stores (D_1, D_2). A very important property of *well-formed* trees is that their equivalence can also be stated in terms of paths:

Theorem 1.[1] *Let* $T = (V, E, L, D)$ *and* $T' = (V', E', L', D')$ *be two well-formed hierarchical data trees. Then,* $T \equiv T'$ *if and only if* $\text{paths}(T) = \text{paths}(T')$.

This theorem states that a given set of paths uniquely defines a well-formed HDT. This property is very important for our approach since our synthesized programs construct the output tree from a set of paths. However, as illustrated by the following example, this property does not hold if we lift the well-formedness assumption:

Example 3. Consider the HDTs of Figure 4, where letters indicate node labels, and assume all nodes store data d. In this case, we have *paths*$(T_1) =$ *paths*(T_2), but the left tree T_1 is not well-formed, as A has two children with label B.

4. Synthesizing Trees from Paths

In this section, we describe our algorithm for synthesizing HDT transformations given an appropriate path transformer.

Overview. The high-level structure of our synthesis algorithm consists of four steps and is summarized in Figure 5. First, given a set of input-output HDTs τ, we verify that examples τ obey a certain *unambiguity* restriction required by our algorithm and enforced using the CHECKEXAMPLES function at line 4. Next, we *furcate* the input-output trees τ into a set $\mathcal{E}$ of *path transformation examples*. Specifically, each example $e \in \mathcal{E}$ maps a path p in input tree T to a "corresponding" path p' in T' for some $(T, T') \in \tau$. Next, we invoke a function called INFERPATHTRANS to learn an appropriate path transformer f such that $p' \in f(p)$ for every

[1] Proofs of all non-trivial statements are given in the appendix.

```
1: procedure SYNTHESIZE(set ⟨Tree, Tree⟩ τ)
2:     Input: Examples τ, consisting of pairs of HDTs
3:     Output: Synthesized program P
4:     if (!CHECKEXAMPLES(τ)) then return ⊥;
5:     E := FURCATE(τ);
6:     f := INFERPATHTRANS(E);
7:     P := CODEGEN(f);
8:     return P;
```

Figure 5. High-level structure of our synthesis algorithm

$(p, p') \in \mathcal{E}$. Finally, CODEGEN generates a program that performs the desired tree transformation by applying f to each path in the input tree and then constructing the output tree from the new set of paths. In what follows, we explain these steps in more detail, leaving the INFERPATHTRANS procedure to Section 5.

Requirements on examples. Our approach is parametric on a notion of *correspondence* between paths in the input and output trees. Let Π be the universe of paths in all possible HDTs. A correspondence relation is a binary relation $\sim \subseteq \Pi \times \Pi$. Given a set of input-output examples τ, let us define $\tau.in, \tau.out$ to be the input and output trees in τ respectively. Our synthesis algorithm expects the user-provided examples τ to obey a certain semantic *unambiguity* criterion:

> ***Unambiguity:*** For every $p' \in$ *paths*$(\tau.out)$, there exists a *unique* p such that $p \sim p'$ where $p \in$ *paths*$(\tau.in)$ and $(p, p') \in$ *paths*$(T) \times$ *paths*(T') for some $(T, T') \in \tau$.

In other words, unambiguity requires that, for every output path p', we can find exactly one input path p such that $p \sim p'$ and p, p' belong to the same input-output example. Unambiguity is enforced by the CHECKEXAMPLES function used at line 4 of the SYNTHESIZE algorithm [2].

The correspondence relation $\sim$ can be defined in many natural ways. Specifically, the HADES system allows the user to mark paths in the input-output examples as corresponding. However, HADES also comes with a default definition of $\sim$ that is adequate in many practical settings.

Furcation. Given an unambiguous set of examples τ, our algorithm *furcates* them into a set of *path transformation examples* $\mathcal{E}$. Specifically, a pair of paths $(p, p') \in \mathcal{E}$ iff $p \in$ *paths*(T), $p' \in$ *paths*(T'), and $p \sim p'$ for some $(T, T') \in \tau$. If some input path $p \in$ *paths*(T) does not have a corresponding output path, then $\mathcal{E}$ also contains $(p, \perp)$.

Note that $\mathcal{E}$ may contain multiple examples that have the same path p as an input. For instance, when some leaf in the input tree has been duplicated in the output tree, then there will be at least two examples (p, p') and (p, p'') in

[2] We can actually drop the unambiguity requirement by adding another layer of search to the synthesis algorithm. However, we have not encountered any examples that violate this restriction in practice.

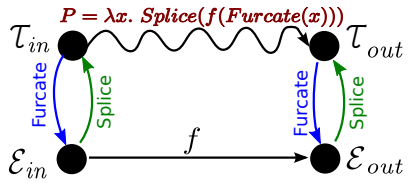


Figure 6. Schematic illustration of our approach.

$\mathcal{E}$. However, due to the unambiguity requirement, it is not possible that there are multiple examples in $\mathcal{E}$ that have the same output path. That is, if $(p, p') \in \mathcal{E}$, then there does not exist another example $(p'', p') \in \mathcal{E}$ where $p \neq p''$.

Given path transformation examples $\mathcal{E}$, we write *inputs*$(\mathcal{E})$ and *outputs*$(\mathcal{E})$ to denote the input and output paths in $\mathcal{E}$ respectively. That is, $p \in$ *inputs*$(\mathcal{E})$ iff $(p, \_) \in \mathcal{E}$.

Example 4. Figure 2 shows the result of furcating the input-output example from Figure 1.

Path transformers. The next step in our synthesis algorithm is to learn a *path transformer* that takes an input path and returns a *set* of output paths. Any path transformer f returned by INFERPATHTRANS at line 6 of the SYNTHESIZE procedure must satisfy the following requirement:

$$\forall p \in \textit{inputs}(\mathcal{E}).\ (p' \in f(p) \Leftrightarrow (p, p') \in \mathcal{E})$$

When an input path p does not have a corresponding output path p', we require that $f(p) = \{\bot\}$. Since INFERPATHTRANS is the most involved aspect of the synthesis algorithm, we discuss it in detail in Section 5.

Code generation. Once we learn a path transformer f, the last step of our algorithm is to generate code for the synthesized program. For this purpose, we first define a *splicing operation*: Given a set S of paths, SPLICE(S) yields a well-formed tree T such that *paths*$(T) = S$. Recall from Theorem 1 that the result of splicing is unique. Using this splicing operation, the CODEGEN procedure used at line 7 of Figure 5 yields the following function P:

$$P = \lambda T.\ \textsc{Splice}(\{p' \mid p' \in f(p) \wedge p \in \textit{paths}(T) \wedge p' \neq \bot\})$$

In other words, synthesized program P constructs the output tree by applying function f to each path in the input tree.

Summary. Figure 6 gives a schematic summary our approach: Our *synthesis algorithm* furcates the input-output examples and learns a path transformer f. On the other hand, the *synthesized algorithm* furcates the input tree, applies path transformer f, and splices it back to obtain the output tree.

Theorem 2. (Soundness) *Let τ be a set of examples satisfying the unambiguity requirement, and let $\mathcal{E}$ be the output of* FURCATE*(τ). Then, $\forall (T, T') \in \tau.\ P(T) \equiv T'$ where P is the output of procedure* SYNTHESIZE *from Figure 5.*

Limitations. Since our approach reduces the synthesis of tree transformations to the problem of synthesizing path transformations, not every tree transformation can be synthesized by our approach. In particular, we can only synthesize tree transformations that are expressible as a combination of *independent* path transformations. As we discuss in Section 5, there are also some restrictions on the class of path transformers that we can generate. Specifically, the path transformers cannot be stateful and must be expressible using mappers over different path segments.

Path transformer π	:=	$\lambda x.\ \{\phi_1 \rightarrow \chi_1 \oplus \ldots \oplus \phi_n \rightarrow \chi_n\}$
Path term χ	:=	$\text{concat}(\tau_1(x), \ldots, \tau_n(x))$
Segment trans. τ	:=	$\lambda x.\ \text{map}\ F\ \text{subpath}(x, t_1, t_2)$
Index term t	:=	$b \cdot \text{size}(x) + c$
Path cond ϕ	:=	$P_i(x) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi$
Mapper F	:=	$\lambda x.\ \text{int} \mid \lambda x.\ f_i(x)$
		$\mid \lambda x.\ \text{if}(\varphi_i(x))\ \text{then}\ f_i(x)\ \text{else}\ f_j(x)$

Figure 7. Language for expressing path transformers.

5. Synthesizing Path Transformations

We now describe the INFERPATHTRANS algorithm for learning path transformers from path examples $\mathcal{E}$. Each example $(p, p') \in \mathcal{E}$ consists of an input path p and an output path p', and our goal is to learn a path transformer f satisfying the property $\forall p \in \textit{inputs}(\mathcal{E}).(p' \in f(p) \Leftrightarrow (p, p') \in \mathcal{E})$.

Language. We first introduce a small language over which we describe path transformers. As shown in Figure 7, a path transformer π takes as input a path x and returns a set of paths $\Pi = \{p_1, \ldots, p_k\}$. Specifically, a path transformer π has the syntax $\lambda x.\ \{\phi_1 \rightarrow \chi_1 \oplus \ldots \oplus \phi_n \rightarrow \chi_n\}$ where each ϕ_i is a *path condition* that evaluates to true or false, and each χ_i is a *path term* describing an output path. The semantics of this construct is that $\chi_i \in \Pi$ iff ϕ_i evaluates to true. We refer to the number of (ϕ_i, χ_i) pairs in π as the *arity* of π.

Path terms χ used in the path transformer are formed by concatanating different subpaths $\tau_i(x)$ where each τ_i is a so-called *segment transformer*. A segment transformer τ is of the form $\lambda x.\text{map}\ F\ \text{subpath}(x, t_1, t_2)$ and applies function F to a subpath of x starting at index t_1 and ending at index t_2 (inclusive). For brevity, we often abbreviate $\lambda x.\text{map}\ F\ \text{subpath}(x, t_1, t_2)$ using the notation $\langle t_1, t_2, F\rangle$. Note that a segment transformer is a kind of looping construct that iterates over a consecutive range of elements in x. Indices in segment transformers are specified using *index terms* t of the form $b \cdot \text{size}(x) + c$ where b is either 0 or 1, c is an integer, and $\text{size}(x)$ denotes the number of elements in path x. [3] Mapper functions F either return constants or apply pre-defined functions f_i to their input. For instance, in the file system domain, such predefined functions include procedures for changing file permission or converting one file type to another (e.g., jpg to png). Mapper functions F can

[3] Our implementation also allows terms containing $\text{indexOf}(x, e)$ expressions; however, we ignore them here to simplify the presentation.

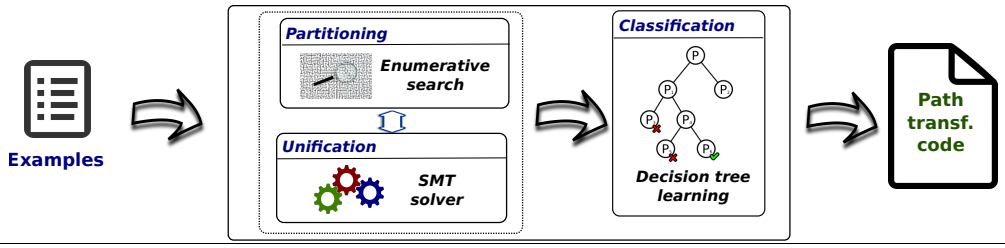


Figure 8. Schematic overview of algorithm for learning path transformers

also contain if statements $\text{if}(\varphi_i(x))$ then $f_i(x)$ else $f_j(x)$ where each φ_i is drawn from a family of pre-defined predicate templates (e.g., for checking file type).

Overview. We now give an overview of our algorithm for learning path transformers. As illustrated in Figure 8, our algorithm consists of three key components, namely *partitioning*, *unification*, and *classification*. The goal of partitioning is to divide examples $\mathcal{E}$ into groups of *unifiable subsets*. We say that a set of examples $\mathcal{E}^*$ is *unifiable* if *outputs*$(\mathcal{E}^*)$ can be represented using the same path term χ^*, and we refer to χ^* as the *unifier* for $\mathcal{E}^*$. Our algorithm represents each partition $\mathcal{P}_i$ as a triple $\langle \mathcal{E}_i, \chi_i, \phi_i \rangle$ where $\mathcal{E}_i$ is a unifiable set of examples, χ_i is their unifier, and ϕ_i is a predicate distinguishing $\mathcal{E}_i$ from the other examples.

The partitioning component of our algorithm is based on enumerative search that tries different hypotheses in increasing order of complexity. Here, a *hypothesis* corresponds to a partitioning of examples $\mathcal{E}$ into k disjoint groups $\mathcal{E}_1, \ldots, \mathcal{E}_k$. Given a hypothesis, we query whether each $\mathcal{E}_i$ is unifiable. If unification fails, we backtrack and try a different hypothesis.

Since our method repeatedly invokes the unification algorithm to confirm or refute a hypothesis, we need an efficient mechanism for finding unifiers. Towards this goal, our algorithm represents each input-output example using a compact numeric representation and invokes an SMT solver to determine the existence of a unifier. Furthermore, we can obtain the unifier χ_i associated with examples $\mathcal{E}_i$ by getting a satisfying assignment to an SMT formula. This approach allows our algorithm to find unifiers with a single SMT query rather than explicitly exploring search spaces of exponential size.

The last key ingredient of our algorithm for synthesizing path transformers is *classification*. Given a set of examples $\mathcal{E}_1, \ldots, \mathcal{E}_k$, the goal of classification is to infer a predicate ϕ_i for each $\mathcal{E}_i$ such that ϕ_i evaluates to *true* for each $p \in$ *inputs*$(\mathcal{E}_i)$ and evaluates to *false* for each $p' \in$ *inputs*$(\mathcal{E}) -$ *inputs*$(\mathcal{E}_i)$. For this purpose, we use the ID3 algorithm for learning a small decision tree and then extract a formula describing all positive examples in this tree.

InferPathTrans algorithm. Figure 9 presents the INFERPATHTRANS procedure based on this discussion. The algorithm consists of two phases: In the first phase, we partition examples $\mathcal{E}$ into a *smallest* set Φ of unifiable groups, and, in the second phase, we infer classifiers for each partition.

```
1:  procedure InferPathTrans(set E)
2:      Input: A set of path transformation examples E
3:      Output: Synthesized path transformer
4:                          ▷ Phase I: Partition into unifiable subsets
5:      for i=1; i≤ |E|; i++  do
6:          Φ := Partition(∅, E, i);
7:          if Φ ≠ ∅ then break;
8:      for all P_i in Φ do          ▷ Phase II: Learn classifiers
9:          P_i.φ := Classify(P_i.E, E);
10:     π := PtCodeGen(Φ);
11:     return π;
```

Figure 9. Algorithm for learning path transformations

Specifically, lines 5-7 try to partition $\mathcal{E}$ into i disjoint groups by invoking the PARTITION procedure, and lines 8-9 infer classifiers. Finally, we use a procedure called PTCODEGEN to generate a path transformer π from the partitions in the expected way. In what follows, we describe partitioning, unification, and classification in more detail.

5.1 Partitioning

Figure 10 shows the partitioning algorithm used in INFERPATHTRANS. The recursive PARTITION procedure takes as input a set of examples $\mathcal{E}_1$ that are part of the same partition, the remaining examples $\mathcal{E}_2$, and number of partitions k. The base case of the algorithm is when $k = 1$: In this case, we try to unify all examples in $\mathcal{E}_1 \cup \mathcal{E}_2$, and, if this is not possible, we return failure (i.e., $\emptyset$).

In the recursive case (lines 9–16), we try to grow the current partition $\mathcal{E}_1$ by adding one or more of the remaining examples from $\mathcal{E}_2$. The algorithm always maintains the invariant that elements in $\mathcal{E}_1$ are unifiable. Hence, we try to add an element $e \in \mathcal{E}_2$ to $\mathcal{E}_1$ (line 10), and if the resulting set is not unifiable, we give up and try a different element (line 11). Since $\mathcal{E}_1 \cup \{e\}$ is unifiable, we now check if it is possible to partition the remaining examples $\mathcal{E}_2 - \{e\}$ into $k - 1$ unifiable sets (recursive call at line 12). If this is indeed possible, we have found a way to partition $\mathcal{E}_1 \cup \mathcal{E}_2$ into k different partitions and return success (line 14).

Now, if the remaining examples $\mathcal{E}_2$ cannot be partitioned into $k - 1$ unifiable sets, we try to shrink $\mathcal{E}_2$ by growing $\mathcal{E}_1$.

```
1:  procedure PARTITION(set E_1, set E_2, int k)
2:      Input: Current partition E_1, remaining examples E_2,
3:             and number of partitions k
4:      Output: Set of partitions Φ
5:      if k = 1 then                                   ▷ Base case
6:          χ := UNIFY(E_1 ∪ E_2);
7:          if χ = null then return ∅;
8:          return {P(E_1 ∪ E_2, χ)};
9:      for all e ∈ E_2 do                              ▷ Recursive case
10:         χ := UNIFY(E_1 ∪ {e});
11:         if χ = null then continue;
12:         Φ := PARTITION(∅, E_2 − {e}, k − 1);
13:         if Φ ≠ ∅ then
14:             return Φ ∪ {P(E_1 ∪ {e}, χ)};
15:         Φ := PARTITION(E_1 ∪ {e}, E_2 − {e}, k);
16:         if Φ ≠ ∅ then return Φ;
17:     return ∅;
```

Figure 10. Partitioning Algorithm. The notation $\mathcal{P}(\mathcal{E}, \chi)$ indicates a partition with examples $\mathcal{E}$ and their unifier χ.

Hence, the recursive call at line 15 looks for a partitioning of examples where one of the partitions contains *at least* $\mathcal{E}_1 \cup \{e\}$. If this recursive call also does not succeed, then we move on and consider the scenario where partition $\mathcal{E}_1$ does not contain the current element e.

Observe that PARTITION$(\emptyset, \mathcal{E}, k)$ effectively explores all possible ways to partition examples $\mathcal{E}$ into k unifiable subsets. However, since most subsets of $\mathcal{E}$ are typically not unifiable, the algorithm does not come anywhere near its worst-case $O(k^n)$ behavior in practice.

5.2 Unification

We now describe the UNIFY procedure for determining if examples $\mathcal{E}$ have a unifier. Since the unification algorithm is invoked many times during partitioning, we need to ensure that UNIFY is efficient in practice. Hence, we formulate it as a symbolic constraint solving problem rather than performing explicit search. However, in order to reduce unification to SMT solving, we first need to represent each input-output example in a so-called *summarized form* that uses a numerical representation to describe each path transformation.

Intuitively, a *summarized example* represents a path transformation as a *permutation* of the elements in the input path. For example, if some element e in the output path has the same label as the k'th element in the input path, then we represent e using numerical value k. On the other hand, if element e does not have a corresponding element with the same label in the input path, summarization uses a so-called *"dictionary"* $\mathcal{D}$ to map e to a numerical value. More formally, we define example summarization as follows:

Definition 5. (Example summarization) Let $\mathcal{E}$ be a set of examples where $\mathcal{L}$ denotes the labels used in $\mathcal{E}$, and let $\mathcal{F}$ be the set of pre-defined functions allowed in the path transformer. Let $\mathcal{D} : (\mathcal{L} \cup \mathcal{F}) \to \{i \mid i \in \mathbb{Z} \wedge i > m\}$ be an injective function where m is the maximum path length in *inputs*$(\mathcal{E})$. Given an example $(p_1, p_2) \in \mathcal{E}$, the summarized form of (p_1, p_2) is a pair (n, σ) where n is the length of path p_1 and σ is a sequence such that:

$$\sigma_i : \begin{cases} (j,\ p_1[j].d \to p_2[i].d) & \textit{if } \exists j.p_2[i].\ell = p_1[j].\ell \\ (\mathcal{D}(f^*) + j,\ \perp \to p_2[i].d) & \textit{else if } \exists j.p_2[i].\ell = f^*(p_1[j]) \\ (\mathcal{D}(p_2[i].\ell),\ \perp \to p_2[i].d) & \textit{otherwise} \end{cases}$$

We illustrate summarization using a few examples:

Example 5. Consider input path $p1 = [(A, r), (B, r), (C, r)]$, and output path $p_2 = [(C, w), (A, r), (B, r)]$, where r, w indicate permissions. The summarized example is $(3, \sigma)$ where $\sigma = [(3, r \mapsto w), (1, r \mapsto r), (2, r \mapsto r)]$. The first element in σ is $(3, r \mapsto w)$ because the first element C of the output path is at index 3 in the input path, and its corresponding data is mapped from r to w.

Example 6. Consider the same p_1 from Example 5 and the output path $p_2' = [(A, r), (B, r), (New, r)]$. Suppose that $\mathcal{D}(New) = 1000$ (i.e., *"dictionary"* assigns 1000 to foreign element *New*). The summarized example is $(3, \sigma')$ where $\sigma' = [(1, r \mapsto r), (2, r \mapsto r), (1000, \perp \mapsto r)]$.

Example 7. Consider the input path $[(A, \perp), (B, pdf)]$ and output $[(A, \perp), (pdf, \perp), (B, pdf)]$. In this case, the summarized example is $(2, \sigma)$ where $\sigma = [(1, \perp \mapsto \perp), (\mathcal{D}(ExtOf) + 2, \perp \mapsto \perp), (2, pdf \mapsto pdf)]$. Note that label *pdf* in the output list is mapped to $\mathcal{D}(ExtOf) + 2$ because it corresponds to the extension for element at index 2 in the input list (case 2 of Definition 5).

Given a summarized example $e = (n, [(i_1, \_), \ldots, (i_n, \_)])$, we write *indices*$(e)$ to denote $[i_1, \ldots, i_n]$. For instance, in Example 5, we have *indices*$(e) = [3, 1, 2]$.

The next step in our unification algorithm is to *coalesce* consecutive indices in the summarized example. Hence, we define the *coalesced form* of an example as follows:

Definition 6. (Coalesced form) Given a summarized example $e = (n, \sigma)$, we say that e^* is a *coalesced form* of e iff it is of the form $(n, [\langle b_1, e_1, M_1\rangle, \ldots, \langle b_k, e_k, M_k\rangle])$ where

- *indices*$(e) = [b_1, \ldots, e_1, \ldots, b_k, \ldots, e_k]$
- $\forall i$, $[b_i, b_{i+1}, \ldots, e_i]$ is a contiguous sublist of *indices*(e)
- $M_k = \bigcup_j \{m_j \mid b_k \le i_j \le e_k \wedge \sigma_j = (i_j, m_j)\}$

Intuitively, this definition "coalesces" consecutive indices in a summarized example. Note that the coalesced form of an example is *not* unique because we are *allowed* but *not required* to coalesce consecutive indices.

Example 8. Consider the summarized example from Example 5, which has the following two coalesced forms:

$$e_1^* = (3, [\langle 3, 3, \{r \mapsto w\}\rangle, \langle 1, 1, \{r \mapsto r\}\rangle, \langle 2, 2, \{r \mapsto r\}\rangle])$$
$$e_2^* = (3, [\langle 3, 3, \{r \mapsto w\}\rangle, \langle 1, 2, \{r \mapsto r\}\rangle])$$

Given a coalesced example $e^* = (n, \sigma^*)$ where σ^* is $[\langle b_1, e_1, M_1\rangle, \ldots, \langle b_k, e_k, M_k\rangle]$, we define *len*$(e^*)$ to be n and *segments*(e^*) to be k. We also write *begin*(e^*, j) to indicate b_j, *end*(e^*, j) for e_j, and *data*(e^*, j) for M_j. Note that σ^* can be viewed as a concatanation of *concrete path segments* of the form $\langle c, c', M\rangle$ where c and c' are the start and end indices for the corresponding path segment respectively.

Before we continue, let us notice the similarity between segment transformers $\langle t, t', F\rangle$ [4] in the language from Figure 7, and each concrete path segment $\langle c, c', M\rangle$ in a coalesced example. Specifically, observe that a concrete path segment can be viewed as a *concrete instantiation* of a segment transformer $\langle b * \mathit{size}(x) + c, b' * \mathit{size}(x) + c', F\rangle$ where each of the terms b, c, b', c', and F are substituted by concrete values. In fact, this is no coincidence: *The key insight underlying our unification algorithm is to use the concrete path segments in the coalesced examples to solve for the unknown terms in segment transformers using an SMT solver.*

Let us now consider the unification algorithm presented in Figure 11. Given examples $\mathcal{E}$, the UNIFY algorithm first computes the summarized examples $\mathcal{E}'$ and then generates all possible coalesced forms (lines 5-6). Since we do not know which coalesced form is the "right" one, we need to consider all possible combinations of coalesced forms of the examples. Hence, set Λ from line 6 corresponds to the Cartesian product of the coalesced form of examples $\mathcal{E}'$.

Next, the algorithm enumerates all possible candidate unifiers χ of increasing arity. Based on the grammar of our language (recall Figure 7), a path term χ of arity k has the shape $\mathrm{concat}(\tau_1(x), \ldots, \tau_k(x))$ where each τ_i is a segment transformer of the form $\langle b_i \cdot \mathit{size}(x) + c_i\ , b_i' \cdot \mathit{size}(x) + c_i'\ , F_i\rangle$. Hence, the hypothesis χ at line 10 is a templatized unifier whose unknown coefficients will be inferred later.

Given a hypothesis χ, we next try to confirm or refute this hypothesis by checking if there exists some $\mathcal{E}^* \in \Lambda$ for which χ is a unifier. For χ to be a unifier for $\mathcal{E}^*$, every example $e_i^* \in \mathcal{E}^*$ must contain exactly k segments because χ has arity k. If this condition is not met (line 13), χ cannot be a unifier for $\mathcal{E}^*$, so we reject it.

If all examples in $\mathcal{E}^*$ contain k segments, we try to instantiate the unknown coefficients $b_1, b_1', c_1, c_1', \ldots, b_k, b_k', c_k, c_k'$ in χ in a way that is consistent with the concrete path segments in all examples in $\mathcal{E}^*$. Now, consider the i'th concrete path segment in coalesced example e^* and the i'th abstract path segment $\langle b_i \cdot \mathit{size}(x) + c_i, b_i' \cdot \mathit{size}(x) + c_i', F_i\rangle$ in hypothesis χ. Clearly, if our hypothesis is correct, it should be possible to instantiate the unknown coefficients in a way that

[4] Recall that $\langle t, t', F\rangle$ is an abbreviation for $\lambda x.\ \mathrm{map}\ F\ \mathrm{subpath}(x, t, t')$.

```
1:  procedure UNIFY(set E)
2:      Input: A set of path transformation examples E
3:      Output: A unifier χ if it exists, null otherwise
4:                          ▷ Convert examples to coalesced form
5:      E' := {e' | e' = SUMMARIZE(e) ∧ e ∈ E};
6:      Λ := {(e*_1, ..., e*_n) | e*_i ∈ COALESCE(e'_i) ∧ e'_i ∈ E'};
7:        ▷ Generate candidate unifiers χ of increasing size
8:      for k=1 to maxSize(outputs(E)) do
9:          τ_i := ⟨b_i · size(x) + c_i , b'_i · size(x) + c'_i , F_i⟩;
10:         χ := concat(τ_1(x), ..., τ_k(x));
11:                       ▷ Check if χ unifies some E* ∈ Λ
12:         for all E* ∈ Λ do
13:             if (∃e*_i ∈ E*. segments(e*_i) ≠ k) then
14:                 continue;
15:               ▷ Use SMT solver to check if χ unifies E*
16:             φ^i_{e*} := (b_i · len(e*) + c_i = begin(e*, i));
17:             ψ^i_{e*} := (b'_i · len(e*) + c'_i = end(e*, i));
18:             φ := ⋀_{1≤i≤k} ⋀_{e*∈E*} (φ^i_{e*} ∧ ψ^i_{e*});
19:             if UNSAT(φ) then continue;
20:             σ := SATASSIGN(φ);
21:             σ' := UNIFYMAPPERS(E*);
22:             if σ' = null then continue;
23:             return SUBSTITUTE(χ, σ ∪ σ');
24:     return null;
```

Figure 11. Unification algorithm

satisfies:

$$b_i \cdot \mathit{len}(e^*) + c_i = \mathit{begin}(e^*, i) \wedge b_i' \cdot \mathit{len}(e^*) + c_i' = \mathit{end}(e^*, i)$$

since the size of this example is *len*(e^*) and begin and end indices for the path segment are *begin*(e^*, i) and *end*(e^*, i). Hence, we test the correctness of hypothesis χ for $\mathcal{E}^*$ by querying the satisfiability of formula ϕ from line 18. If ϕ is unsatisfiable (line 19), we reject the hypothesis for $\mathcal{E}^*$.

If, however, ϕ is satisfiable, we have found an instantiation of the unknown coefficients, which is given by the satisfying assignment σ at line 20. Now, the only remaining question is whether we can also find an instantiation of the unknown functions $F_1, \ldots, F_k$ used in χ. For this purpose, we use a function called UNIFYMAPPERS which tries to find mapper functions F_i unifying all the different M_i's from the examples. Since the UNIFYMAPPERS procedure is based on straightforward enumerative search, we do not describe it in detail. In particular, since the language of Figure 7 only allows a finite set of pre-defined data transformers f_i and predicates ϕ_i, UNIFYMAPPERS enumerates –in increasing order of complexity– all possible functions belonging to the grammar of mapper functions in Figure 7.

$$
\begin{array}{ll}
\mathcal{E}_1': & (3, [(1, d_m \mapsto d_m), (\mathcal{D}(\textit{ExtOf}) + 3, \bot \mapsto d_f), (2, d_j \mapsto d_j), (3, d_n \mapsto d_n)]) \\
\mathcal{E}_3': & (3, [(1, d_m \mapsto d_m), (\mathcal{D}(\textit{ExtOf}) + 3, \bot \mapsto d_o), (2, d_r \mapsto d_r), (3, d_h \mapsto d_h)]) \\
\mathcal{E}_4': & (4, [(1, d_m \mapsto d_m), (\mathcal{D}(\textit{ExtOf}) + 4, \bot \mapsto d_f), (2, d_r \mapsto d_r), (3, d_{ms} \mapsto d_{ms}), (4, d_b \mapsto d_b)]) \\
\mathcal{E}_6': & (4, [(1, d_m \mapsto d_m), (\mathcal{D}(\textit{ExtOf}) + 4, \bot \mapsto d_{mp}), (2, d_p \mapsto d_p), (3, d_a \mapsto d_a), (4, d_t \mapsto d_t)])
\end{array}
$$

Figure 12. Summarization of partition $\mathcal{P}_1$ of the motivating example.

$$
\begin{array}{ll}
\mathcal{E}_1^*: & (3, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\textit{ExtOf}) + 3, \mathcal{D}(\textit{ExtOf}) + 3, \{\bot \mapsto d_f\}\rangle, \langle 2, 3, \{d_j \mapsto d_j, d_n \mapsto d_n\}\rangle]) \\
\mathcal{E}_3^*: & (3, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\textit{ExtOf}) + 3, \mathcal{D}(\textit{ExtOf}) + 3, \{\bot \mapsto d_o\}\rangle, \langle 2, 3, \{d_r \mapsto d_r, d_h \mapsto d_h\}\rangle]) \\
\mathcal{E}_4^*: & (4, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\textit{ExtOf}) + 4, \mathcal{D}(\textit{ExtOf}) + 4, \{\bot \mapsto d_f\}\rangle, \langle 2, 4, \{d_r \mapsto d_r, d_{ms} \mapsto d_{ms}, d_b \mapsto d_b\}\rangle]) \\
\mathcal{E}_6^*: & (4, [\langle 1, 1, \{d_m \mapsto d_m\}\rangle, \langle \mathcal{D}(\textit{ExtOf}) + 4, \mathcal{D}(\textit{ExtOf}) + 4, \{\bot \mapsto d_{mp}\}\rangle, \langle 2, 4, \{d_p \mapsto d_p, d_a \mapsto d_a, d_t \mapsto d_t\}\rangle])
\end{array}
$$

Figure 13. A set of coalesced examples $\mathcal{E}^*$ for partition $\mathcal{P}_1$

$$
\begin{array}{ll}
\phi: & b_1 \cdot 3 + c_1 = 1 \ \wedge\ b_1' \cdot 3 + c_1' = 1 \ \wedge\ b_1 \cdot 4 + c_1 = 1 \ \wedge\ b_1' \cdot 4 + c_1' = 1 \ \wedge \\
& b_2 \cdot 3 + c_2 = \mathcal{D}(\textit{ExtOf}) + 3 \ \wedge\ b_2' \cdot 3 + c_2' = \mathcal{D}(\textit{ExtOf}) + 3 \ \wedge\ b_2 \cdot 4 + c_2 = \mathcal{D}(\textit{ExtOf}) + 4 \ \wedge\ b_2' \cdot 4 + c_2' = \mathcal{D}(\textit{ExtOf}) + 4 \ \wedge \\
& b_3 \cdot 3 + c_3 = 2 \ \wedge\ b_3' \cdot 3 + c_3' = 3 \ \wedge\ b_3 \cdot 4 + c_3 = 2 \ \wedge\ b_3' \cdot 4 + c_3' = 4
\end{array}
$$

Figure 14. (Simplified) formula ϕ to check the satisfiability of hypothesis χ_1 on set $\mathcal{E}^*$.

Example 9. For the motivating example from Section 2, our unification algorithm takes the following steps to determine unifier χ_1 for partition $\mathcal{P}_1$: First, we generate the summarized examples shown in Figure 12 and construct set Λ. We then consider hypotheses of increasing size and reject those with arity 1 and 2 since all examples contain at least 3 segments. Now, let's consider hypothesis χ of arity 3 and the set of coalesced examples $\mathcal{E}^*$ shown in Figure 13. We generate the formula ϕ shown in Figure 14 and get a satisfying assignment, which results in the following instantiation of χ:

$$[\langle 1, 1, F_1\rangle, \langle v, v, F_2\rangle, \langle 2, \textit{size}(x), F_3\rangle]$$

where $v = \mathcal{D}(\textit{ExtOf}) + \textit{size}(x)$. Finally, UNIFYMAPPERS searches for instantiations of F_1, F_2, and F_3 satisfying all data mappers in the examples. For F_1 and F_3, it returns the *Identity* function, and for, F_2, it extracts the function *extOf* from the segment transformer coefficients. As a result, we obtain the unifier χ_1 from Section 2.

5.3 Classification

We now consider the last missing piece of our algorithm, namely classification. Given examples $\mathcal{E}$ and partition $\mathcal{P}_i$ with examples $\mathcal{E}_i \subseteq \mathcal{E}$ and unifier χ_i, the goal of classification is to find a predicate ϕ_i such that:

(1) $\forall p \in \textit{inputs}(\mathcal{E}_i).\ (\phi_i[p/x] \equiv \textit{true})$
(2) $\forall p \in (\textit{inputs}(\mathcal{E}) - \textit{inputs}(\mathcal{E}_i)).\ (\phi_i[p/x] \equiv \textit{false})$

Our key insight is that the inference of such a predicate ϕ_i is precisely the familiar classification problem in machine learning. Hence, to find predicate ϕ_i, we first extract relevant features from each path and then use decision tree learning.

Feature extraction. To use decision tree learning for classification, we need to represent each input path using a finite set of discrete features. In the HADES system, these features are domain-specific and therefore defined separately for each application domain. For instance, *some* of the features for the file system domain include file types, permissions, and the presence of a certain file or directory in the path. Given path p, we write $\alpha(p)$ to denote the feature vector for p and $\alpha_f(p)$ for the value of feature f for path p.

Decision tree learning. We now explain how to use decision tree learning to infer a predicate distinguishing paths Π_1 from those in Π_2. Given sets Π_1 and Π_2 and a set of features $\mathcal{F}$, we use the ID3 algorithm [20] to construct a decision tree $\mathcal{T}_\mathcal{D}$ with the following properties:

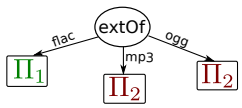
- Each leaf in $\mathcal{T}_\mathcal{D}$ is labeled as Π_1 or Π_2
- Each internal node of $\mathcal{T}_\mathcal{D}$ is labeled with a feature $f \in \mathcal{F}$
- Each edge (f, f', ℓ) from f to f' is annotated with a label ℓ that indicates a possible value of feature f
- Let $(f_1, f_2, \ell_1), \ldots (f_n, \Pi_i, \ell_n)$ be a root-to-leaf path in $\mathcal{T}_\mathcal{D}$. Then, for every $p \in \Pi_1 \cup \Pi_2$, we have:

$$(\bigwedge_{i=1}^{n} \alpha_{f_i}(p) = l_i) \Leftrightarrow p \in \Pi_i$$

Given such a decision tree $\mathcal{T}_\mathcal{D}$, identifying a predicate ϕ differentiating Π_1 from Π_2 is simple. Let $\pi = \langle (f_1, f_2, \ell_1), \ldots (f_n, \Pi_1, \ell_n)\rangle$ be a path in $\mathcal{T}_\mathcal{D}$, and let $\varphi(\pi)$ denote the formula $\wedge_{i=1}^{n}(f_i = l_i)$. Assuming Π_1 corresponds to $\textit{inputs}(\mathcal{E}_i)$ and Π_2 is $\textit{inputs}(\mathcal{E}) - \textit{inputs}(\mathcal{E}_i)$, the following DNF formula ϕ_i gives us a classifier for $\mathcal{P}_i$:

$$\phi_i: \bigvee_{\pi \in \textit{pathTo}(\mathcal{T}_\mathcal{D}, \Pi_1)} \varphi(\pi)$$

Example 10. Consider partition $\mathcal{P}_2$ from the example of Section 2. Here, $\Pi_1 = \{p_1, p_3\}$ and $\Pi_2 = \{p_2, p_4\}$. After running ID3, we obtain the following decision tree:



Hence, we extract the classifier ϕ_2 : *ExtOf(x) = flac*.

6. Implementation

We have implemented our synthesis algorithm in a tool called HADES, which consists of $\approx 9,500$ lines of C++ code.The only external tool used by HADES is the Z3 SMT solver [7]. The core of HADES is the domain-agnostic synthesis backend, which accepts input-output examples in the

form of hierarchical data trees and emits path transformation functions in the intermediate language of Figure 7.

HADES provides an interface for domain-specific plug-ins, and our current implementation incorporates two such front ends: one for XML transformations using XSLT and another one for bash scripts. However, HADES can be extended to new domains by implementing plug-ins that implement the following functionality: (a) represent input-output examples as HDTs; (b) use the synthesized path transformer to emit tree transformation code in the target language; (c) specify any domain-specific functions and features.

7. Evaluation

We evaluated HADES by using it to automate 36 data transformation tasks in the file system and XML domains. Our examples come from two sources: on-line forums (e.g., Stackoverflow, bashscript.org) and teaching assistants at our institution. To simulate a real-word usage scenario of HADES where end-users provide input-output examples, we performed a user study involving six students, only three of which are CS majors. The students in our study neither had prior knowledge of HADES nor are they familiar with program synthesis research. We randomly assigned 6 benchmarks to each participant, and each benchmark was assigned to only one participant.

Prior to the evaluation, we gave the participants a demo of the system, explained how to provide input/output examples, and how to check whether the generated script is correct. For each benchmark, we provided the users an English description of the task to be performed as well as a set of test cases to assess whether HADES produces the correct result. The participants were asked to come up with a set of examples different from the given tests for each benchmark, and then run the tests to verify whether HADES' output is correct. If HADES failed to produce the correct result, the users were asked to restart the process with a modified set of input-output examples. We checked all examples provided by the participants to ensure they did not use any of tests as input to the system.

Figure 15 summarizes the results of our evaluation. The column labeled "Description" provides a brief summary of each benchmark, and "Time" reports synthesis time in seconds. The column labeled "Script" gives statistics about the synthesized script, while the column named "User" provides important data related to user interaction.

Performance. To evaluate performance, we utilized the examples provided by the participants from our user study. [5] All performance experiments are conducted on a MacBook Pro with 2.6 GHz Intel Core i5 processor and 8 GB of 1600 MHz DDR3 memory running OS X version 10.10.3.

[5] When there were multiple rounds of interaction with the user, we used the examples from the last round.

The column labeled "Time" in Figure 15 reports the *total* synthesis time in seconds, including conversion of examples to HDTs and emission of bash or XSLT code. On average, HADES takes 0.90 seconds to synthesize a directory transformation and 0.51 seconds to synthesize an XML transformation. Across all benchmarks, HADES is able to synthesize 91.6% of the benchmarks in under 1 second and 97.2% of the benchmarks in under 10 seconds.

Complexity. The column labeled "Script" in Figure 15 reports various statistics about the script synthesized by HADES: "Branches" reports the number of branches in the synthesized program. ,"Segments" reports the number of loops , and "LOC" gives the number of lines of code for the synthesized *path transformer*. Note that the whole script synthesized by HADES is actually significantly larger (due to furcation/splicing code); the statistics here only include the path transformation portion. As summarized by this data, the scripts synthesized by HADES are fairly complex: They contain between 1-4 branches, 1-10 loops, and between 47-517 lines of code for the path transformer. Furthermore, since our algorithm always generates a simplest path transformer, the reported statistics give a lower bound on the complexity of the required path transformations.

Usability. The last part of Figure 15 reports data about our user study: The column "Iteration" reports the number of rounds of tool-user interaction, "Examples" gives the number of furcated examples, and "Depth" indicates the maximum depth of the input-output trees. Our results demonstrate that HADES is user-friendly: 88.8% of the benchmarks require only 1-2 rounds of user interaction, with no task requiring more than 4 rounds. Furthermore, 72.2% of the tasks require less than 5 examples, and no task requires more than 9. Finally, tree depth is typically very small – by providing example trees with depth 3.2 on average, users are able to obtain scripts that work on trees of unbounded depth.

Comparison with other tools. To substantiate our claim that our approach broadens the scope of tree transformations that can be automatically synthesized, we also compared HADES with two existing tools, namely λ^2 [10] and Myth [18], for synthesizing higher-order functional programs. Since these tools do not directly handle Unix directories or XML documents, we manually created a simplified tree abstraction for each task and supplied these tools with a suitable set of input-output examples. Myth was unable to synthesize any of our benchmarks, and λ^2 was only able to synthesize a single example (X3) within a time limit of 600 seconds. Since these tools target a much broader class of synthesis tasks, their search space seems to blow up when presented with non-trivial tree-transformation tasks. In contrast, by learning path transformers that are applied to each path in the tree, our synthesis algorithm can synthesize complex transformations in a practical manner.

Benchmarks			Time	Script			User		
		Description	Total (s)	Branches	Segments	LOC	Iterations	Examples	Depth
File System	F1	Categorize .csv files based on their group	0.03	1	2	47	2	2	3
	F2	Make all script files executable	0.01	2	2	51	1	2	2
	F3	Copy all text and bash files to directory temp	0.05	2	3	56	2	4	3
	F4	Append last 3 directory names to file name and delete directories	0.02	1	2	50	2	2	6
	F5	Put files in directories based on modification year/month/day	0.02	1	4	52	1	2	4
	F6	Copy files without extension into the “NoExtension” directory	0.05	2	3	56	2	7	4
	F7	Archive each directory to a tarball with modify month in its name	0.01	1	1	50	1	2	2
	F8	Make files in “DoNotModify” directory read-only	0.12	2	2	83	2	4	3
	F9	Convert .mp3, .wma, and .m4a files to .ogg	0.02	1	1	47	1	3	3
	F10	Change group of text files in “Public” directory to “everyone”	0.06	2	2	77	1	3	2
	F11	Change directory structure	0.03	1	4	52	2	3	6
	F12	Convert .zip archives to tarballs	0.08	2	2	77	1	3	3
	F13	Organize all files based on their extensions	1.85	4	10	148	1	9	3
	F14	Append modification date to the file name	0.01	1	2	48	1	2	3
	F15	Convert pdf files to swf files	0.03	2	2	55	2	2	2
	F16	Delete files which are not modified last month	0.03	2	2	54	2	5	2
	F17	Convert video files to audio files and put them in “Audio” directory	0.01	1	2	49	1	3	5
	F18	Convert xml files $\geq$ 1kB to text files	0.09	2	2	77	2	4	2
	F19	Append “lgst” to name of largest file and “sml” to xml files $\leq$ .1kB	17.94	3	5	110	2	6	3
	F20	Extract tarballs to a directory named using file and parent directory	0.36	2	3	83	2	3	3
	F21	Append parent name to each .c file and copy under “MOSS”	0.04	2	3	56	2	4	4
	F22	Keep all files older than 5 days	0.59	2	2	54	3	7	2
	F23	Copy each file to the directory created with its file name	0.02	1	2	47	1	3	5
	F24	Archive directories which are not older than a week	0.11	2	2	80	3	5	2
XML	X1	Add style=’bold’ attribute to parent element of each text	0.02	1	2	145	2	3	5
	X2	Merge elements with “status” attribute and put under children	0.02	1	3	169	3	3	5
	X3	Remove all attributes	0.01	1	1	115	2	3	3
	X4	Change the root element of xml	0.02	1	2	222	1	2	3
	X5	Remove 3rd element and put all nested elements under parent	0.02	1	2	129	1	3	4
	X6	Create a table which maps each text to its parent element tag	0.09	2	10	497	1	6	4
	X7	Remove text in element “done” and put all other text under “todo”	0.05	2	3	241	4	8	3
	X8	Generate HTML drop down list from a XML list storing the data	0.02	1	4	443	1	3	3
	X9	Rename a set of element tags to standard HTML tags	0.02	1	4	356	1	2	3
	X10	Move “class” attribute and categorize based on “class”	0.18	2	6	229	1	6	3
	X11	Categorize based on “tag” and put each class under element with valid HTML tag	5.55	3	9	517	2	6	3
	X12	Delete all elements with tag “p”	0.05	2	1	122	1	5	4

Figure 15. File System and XML Benchmarks

8. Related work

Program synthesis has recently received much attention in the programming languages community. While many approaches require a programmer-provided “template” [4, 24, 25] or a complete logical specification [16], our method performs synthesis from examples.

There is a growing literature on example-driven synthesis. Many of these approaches focus on non-hierarchical data like numbers, strings, and tables [3, 12, 14, 19, 23]. However, several recent efforts study the synthesis of programs over recursive data structures [2, 10, 15, 17, 18]. For example, FlashExtract synthesizes programs made from higher-order combinators using a custom deductive procedure [17], and Escher uses goal-directed enumerative search to synthesize first-order programs with recursive functions [2]. Similarly, λ^2 synthesizes higher-order functional programs using a combination of deduction and cost-directed enumeration [10]. Osera et al. study a similar problem and offer a solution based on type-directed deduction [18].

Unlike the above approaches, our algorithm is based on a reduction of synthesis over trees to synthesis over lists, which is performed using SMT solving and decision tree learning. Furthermore, since we specifically target hierarchical data transformations (rather than synthesis of arbitrary programs), this tighter focus allows us handle tree transformation benchmarks whose complexity exceeds those in prior work. For example, unlike prior efforts, our method is able to

synthesize transformations that alter the hierarchical structure of an input tree. So far as we know (and as demonstrated empirically in Section 7), such benchmarks fall outside the scope of prior approaches to example-driven synthesis.

Our subroutine for synthesizing path transformations bears similarities to FlashFill's strategy for synthesizing string transformations [12]. Similar to our INFERPATHTRANS procedure, FlashFill uses a combination of partitioning and unification; but the algorithmic details are very different. For example, FlashFill maintains a DAG representation of all string transformations that fit a set of examples. In contrast, we use a numerical representation of the input-output examples and reduce unification to SMT solving.

The StriSynth tool described in [13] extends FlashFill and uses it to automate certain kinds of file manipulation tasks, such as renaming files and directories. While the StriSynth approach can handle sophisticated transformations involving file names, it does not address transformations that handle the directory structure. In contrast, this paper addresses general tree transformations and can synthesize bash scripts that modify the directory structure.

There is prior work on learning *finite-state* tree transducers from examples [6, 8, 9, 21]. Our approach is more general than these approaches, in that we handle trees whose nodes contain data of arbitrary types and do not impose a priori limitations on changes to paths in an input tree. However, this generality comes at a cost: our approach lacks the crisp complexity guarantees that some of these automata-theoretic algorithms possess [8].

Decision trees are a popular data structure in machine learning and data mining. They have also found applications in program analysis, for example in precondition learning for procedures [22], identification of latent code errors [5], and repair of programs that manipulate relational databases [11]. So far as we know, the only previous use of decision trees in synthesis is for learning `WHERE` clauses in automated synthesis of SQL queries [26].

9. Conclusion

We have presented an algorithm for synthesizing tree transformations from examples. The central idea of our approach is to reduce the generation of tree transformations to the synthesis of list (path) transformations. The path transformations are synthesized using a novel combination of decision tree learning and SMT solving. The reduction from tree to path transformations simplifies the underlying synthesis algorithm, while allowing us to handle a rich class of tree transformations, including those that restructure the tree.

On the practical side, we have shown that our algorithm has numerous applications for automating the manipulation of hierarchically structured data, such as XML files and Unix directories. In the longer run, approaches like ours can be embedded into end-user programming tools such as Apple's Automator [1], which offers visual abstractions for everyday scripting tasks. Since HADES allows users to generate complex programs from simple examples, it offers a plausible way to broaden the scope of such tools.

Acknowledgments

We thank Thomas Dillig, Yu Feng, Xinyu Wang and Jia Chen for their insightful feedback. We also thank all participants in our user study. Finally, we would like to thank the anonymous reviewers for their thorough and helpful comments.

References

[1] Automator. http://automator.us.

[2] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.

[3] D. W. Barowy, S. Gulwani, T. Hart, and B. G. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, pages 218–228, 2015.

[4] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234, 2014.

[5] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.

[6] J. Carme, R. Gilleron, A. Lemay, and J. Niehren. Interactive learning of node selecting tree transducer. *Machine Learning*, 66(1):33–67, 2007.

[7] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] F. Drewes and J. Högberg. Learning a regular tree language from a teacher. In *Developments in Language Theory*, pages 279–291. Springer, 2003.

[9] J. Eisner. Learning non-isomorphic tree mappings for machine translation. In *Proceedings of Meeting on Association for Computational Linguistics*, pages 205–208, 2003.

[10] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239, 2015.

[11] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *ICSE*, pages 243–253. ACM, 2014.

[12] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[13] S. Gulwani, M. Mayer, F. Niksic, and R. Piskac. Strisynth: synthesis for live programming. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 701–704. IEEE Press, 2015.

[14] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328. ACM, 2011.

[15] E. Kitzelmann. Analytical inductive functional programming. In *Logic-Based Program Synthesis and Transformation*, pages 87–102. Springer, 2009.

[16] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[17] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, page 55, 2014.

[18] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630, 2015.

[19] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, page 43, 2014.

[20] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[21] W. C. Rounds. Mappings and grammars on trees. *Theory of Computing Systems*, 4(3):257–287, 1970.

[22] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 295–306. ACM, 2008.

[23] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*, pages 634–651. Springer, 2012.

[24] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[25] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6): 497–518, 2013.

[26] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 224–234. IEEE, 2013.

Appendix

Proof of Theorem 1. Part 1, $\Rightarrow$. Suppose $T \equiv T'$. We show that *paths*(T) = *paths*(T'). Since $T \equiv T'$, we have $\text{height}(T) = \text{height}(T') = h$. Let $v = \text{root}(T)$ and $v' = \text{root}(T')$. Since $T \equiv T'$, we have $L(v) = L(v') = \ell$ and $D(v) = D'(v') = d$. The proof proceeds using induction on h. For the base case, we consider $h = 1$. In this case, $V = \{v\}$, $V' = \{v'\}$, $E = E' = \emptyset$. Hence, *paths*(T) = *paths*(T') $= (\ell, d)$. For the inductive case, let $h = k + 1$ where $k \geq 1$. Let $C = \text{children}(v)$ and let $C' = \text{children}(v')$. Since $T \equiv T'$, $L(C) = L'(C')$, and since T and T' are well-formed, there exists a one-to-one correspondence $f : C \rightarrow C'$ such that: $f(v_c) = v'_c$ iff$L(v_c) = L'(v'_c)$. Using this fact and condition (3) of Definition 4, we know that, for every $v_c \in \text{children}(v) = C$, $\text{subtree}(T, v_c) \equiv \text{subtree}(T', f(v_c))$. Now, observe that:

$$\text{paths}(T) = \bigcup_{v_c \in C} \{((\ell, d), p_i) \mid p_i \in \text{paths}(\text{subtree}(T, v_c))\}$$

and $\text{paths}(T')$ is equal to:

$$\bigcup_{f(v_c) \in C'} \{((\ell, d), p'_i) \mid p'_i \in \text{paths}(\text{subtree}(T', f(v_c)))\}$$

Using the inductive hypothesis, we have $\text{paths}(T) = \text{paths}(T')$.

Part 2, $\Leftarrow$. Suppose *paths*(T) = *paths*(T'), and let $v = \text{root}(T)$ and $v' = \text{root}(T')$ and $L(v) = \ell, L(v') = \ell', D(v) = d, D(v') = d'$. We show that $T \equiv T'$. First, observe that *paths*(T) = *paths*(T') implies $\text{height}(T) = \text{height}(T')$. The proof proceeds by induction on h. When $h = 1$, *paths*$(T) = \{(\ell, d)\}$ and *paths*$(T') = \{(\ell', d')\}$. Since *paths*(T) = *paths*(T'), this implies $\ell = \ell'$ and $d = d'$. Hence, $T \equiv T'$. For the inductive case, let $h = k + 1$ where $k \geq 1$. Let $C = \text{children}(v)$ and let $C' = \text{children}(v')$. Now, we have:

$$\text{paths}(T) = \bigcup_{v_i \in C} \{((\ell, d), p_i) \mid p_i \in \text{paths}(\text{subtree}(T, v_i))\}$$

and

$$\text{paths}(T') = \bigcup_{v'_i \in C'} \{((\ell', d'), p'_i) \mid p'_i \in \text{paths}(\text{subtree}(T', v'_i))\}$$

Since *paths*(T) = *paths*(T'), this implies $\ell = \ell'$ and $d = d'$ and, since T, T' are well-formed, for each $v_i \in C$, there must be a one-to-one correspondence $f : C \rightarrow C'$ such that $v'_i = f(v_i)$ iff $\text{paths}(\text{subtree}(T, v_i)) = \text{paths}(\text{subtree}(T', v'_i))$. Now, for any pair $(v_i, f(v_i))$, we have $L(v_i) = L'(f(v_i))$ because $\text{paths}(\text{subtree}(T, v_i)) = \text{paths}(\text{subtree}(T', v'_i))$. Note that this implies condition (2) of Definition 4. Furthermore, since $\text{paths}(\text{subtree}(T, v_i)) = \text{paths}(\text{subtree}(T', f(v_i)))$, the inductive hypothesis implies $\text{subtree}(T, v_i) \equiv \text{subtree}(T, v'_i)$. Hence, condition (3) of Definition 4 is also satisfied.

Proof of path transformer property. Let $\Phi = \{P_1, \ldots, P_k\}$ be the set of partitions inferred by INFERPATHTRANS at the end of Phase I (see Figure 9), and let each $\mathcal{P}_i$ be the triple $\langle \mathcal{E}_i, \chi_i, \phi_i \rangle$. Then the path transformer f synthesized by INFERPATHTRANS is:

$$\lambda x.\ \{\phi_1 \rightarrow \chi_1 \oplus \ldots \oplus \phi_k \rightarrow \chi_k\}$$

We first prove that $((p, p') \in \mathcal{E}) \Rightarrow (p' \in f(p))$. First, observe that, for any k, if $\text{PARTITION}(\emptyset, \mathcal{E}, k) \neq \emptyset$, we have:

$$(\bigcup_{\mathcal{P}_i \in \Phi} \mathcal{E}_i) = \mathcal{E}$$

Hence, any $(p, p') \in \mathcal{E}$ must belong to the examples of some partition $\mathcal{P}_i$. Furthermore, our classification algorithm guarantees that, for any $p \in$ *inputs*$(\mathcal{E}_i)$, we have $\phi_i[p/x] \equiv$ *true*. Hence, we know that $\chi_i \in f(p)$. Since $\text{UNIFY}(\mathcal{E}_i) = \chi_i \neq \text{null}$, we have $\chi_i[p/x] = p'$. This implies $p' \in f(p)$.

We now prove the other direction of the property, i.e.:

$$((p \in \textit{inputs}(\mathcal{E}) \wedge p' \in f(p)) \Rightarrow (p, p') \in \mathcal{E})$$

Suppose that $p \in$ *inputs*$(\mathcal{E})$ and $p' \in f(p)$. Since $p' \in f(p)$, there must exist some $\mathcal{P}_i = \langle \mathcal{E}_i, \chi_i, \phi_i \rangle$ such that $\phi_i[p/x] \equiv$ *true* and $\chi_i[p/x] = p'$. Recall that classification guarantees:

(a) $\forall p \in$ *inputs*$(\mathcal{E}_i)$. $(\phi_i[p/x] \equiv$ *true*$)$
(b) $\forall p \in ($*inputs*$(\mathcal{E}) -$ *inputs*$(\mathcal{E}_i))$. $(\phi_i[p/x] \equiv$ *false*$)$

Since $p \in$ *inputs*$(\mathcal{E})$, this implies $p \in$ *inputs*$(\mathcal{E}_i)$; otherwise we would have $\phi_i[p/x] \equiv$ *false*. Hence, we must have $(p, p') \in \mathcal{E}_i$, which in turn implies $(p, p') \in \mathcal{E}$.

Proof of Theorem 2. Suppose $P(T) = T^*$. We will show *paths*(T^*) = *paths*(T'), which implies $T^* \equiv T'$ by Theorem 1. First, we show that, if $p' \in$ *paths*(T'), then $p' \in$ *paths*(T^*). By the unambiguity criterion, there exists a unique $p \in$ *paths*(T) such that $p \sim p'$. By the correctness requirement for path transformer f, we know $p' \in f(p)$ and $p' \neq \perp$ since $p' \in$ *paths*(T'). Hence, $p' \in S'$ where $S' = \{p' \mid p' \in f(p) \wedge p \in$ *paths*$(T) \wedge p' \neq \perp\}$. Since $T^* = \text{SPLICE}(S')$ (recall code generation in Section 4) and SPLICE guarantees that *paths*$(T^*) = S'$, we also have $p' \in$ *paths*(T^*). Now, we show that, if $p^* \in$ *paths*(T^*), then $p^* \in$ *paths*(T'). Since $p^* \in$ *paths*(T^*), there must exist a $p \in$ *paths*(T) such that $p^* \in f(p)$. Since $p \in$ *inputs*$(\mathcal{E})$, correctness of f implies there exists some $(p, p^*) \in \mathcal{E}$. Now, suppose $p^* \notin$ *paths*(T'). Since $(p, p^*) \in \mathcal{E}$ but $p^* \notin$ *paths*(T'), there are two possibilities: (i) Either $p^* = \perp$, or (ii) there is some other $(T_1, T_2) \in \tau$ that results in p^* getting added to $\mathcal{E}$. Now, (i) is not possible because *paths*(T^*) cannot contain $\perp$, and (ii) is not possible due to the unambiguity requirement.