Abstract
This paper presents a new approach for automatically synthesizing transformations on hierarchically 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 algorithm for synthesizing HDT transformations from input-output examples. The key insight underlying our algorithm is to reduce the task of synthesizing tree transformations to that of inferring path transformations. We also present a novel algorithm based on SMT solving and decision tree learning for synthesizing path transformers. We have implemented the proposed ideas in a system called HADES and used it for synthesizing transformations on Unix directories and XML documents. Our evaluation shows that HADES is able to automatically synthesize a variety of interesting transformations that we have collected from online forums.

1. Introduction
Much of the data that users deal with today are inherently hierarchical in nature. Some common examples of such hierarchically-structured data include, but are not limited to, the following:

- **File systems**: Modern operating systems represent the data stored on disk 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 kinds of scientific documents are stored as HDF files that follow a tree structure. In this format, so-called groups correspond to internal nodes of the tree, and datasets represent leaves.
- **Hierarchical spreadsheets**: Several varieties of data organization software (e.g., TreeSheets [3], Smartsheet [2]), allow users to construct so-called hierarchical spreadsheets where data is arranged and visualized in a tree-like format.

Given the ubiquity of such hierarchical data, a common problem that end-users face is to perform various kinds of tree transformation tasks on this data. For instance, consider the following motivating scenarios:

- Given a directory tree called Music with subdirectories for different musical genres, a user wants to re-organize her music files so that Music has subdirectories for different classes of files (e.g., mp3, wma), and each such subdirectory has further subdirectories that correspond to genres (Rock, Jazz etc.).
- Given an XML file, a user wants to convert the name attribute associated with a person tag (e.g., `<person name="..."> ... </person>` to a nested element within the person tag (e.g., `<person><name>... </name></person>`).

Since manually performing these kinds of transformations on large datasets is tedious and error-prone, an attractive alternative is to implement a program, such as a bash script or XSLT program, to automate the desired task. Motivated by these scenarios, this paper proposes a new algorithm, and its implementation in a system called HADES\(^1\), 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). The algorithm does not place any restrictions on the depth or fanout of the hierarchy and 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 modifying data and metadata.

From a technical perspective, a key idea underlying our approach is to reduce the problem of synthesizing tree transformations to that of learning path transformations. Specifically, rather than directly synthesizing a tree transformation, our algorithm represents hierarchically structured data as a set of paths and learns a transformation that allows us to convert each path in the input tree to a corresponding path in the output tree. After learning such a path transformation, our approach synthesizes a program that generates all paths in the input tree, applies the transformation to each path, and then reconstructs the corresponding output tree. As a result, the scripts generated by our technique can be more complex than those that operate directly on trees. However, a key advantage of this approach is that it simplifies the synthesis task and allows us to handle a broad class of hierarchical data transformations in a practical manner.

Another important aspect of our approach is a new algorithm for learning path transformations. Given a set of input-output paths, our approach synthesizes a simplest path transformer minimizing the number of branches in the synthesized program. To learn such a path transformer, our approach alternates between two phases called unification and classification. Given a set \(S\) of pairs of input and output paths, the goal of unification is to determine whether there exists a conditional-free program that achieves the desired transformation on \(S\). By contrast, the goal of classification is to learn concise predicates that differentiate one unifiable subset of examples from another. Our new synthesis algorithm reduces unification to an SMT solving problem and uses decision tree learning for performing classification.

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 domain-specific language (DSL), provided it has been plugged into our infrastructure. Our current implementation provides two DSL front-ends, one that generates bash scripts for Unix directories, and another that outputs XSLT [4] code for performing XML transformations.

To summarize, this paper makes the following contributions:

---

\(^1\) stands for HierArchical Data transformation Engine and Synthesizer
We present a method, embodied in a system called HADES, for synthesizing transformations on hierarchically structured data. The class of programs that can be synthesized by our approach are useful to end-users in a variety of different scenarios, such as those involving file systems, XML files, or hierarchical spreadsheets.

We show how to reduce the problem of synthesizing tree transformations to the simpler task of synthesizing path transformations. We also prove the soundness and relative completeness of this approach under certain lightweight restrictions on the input-output trees.

We present a new synthesis algorithm based on SMT solving and decision tree learning for synthesizing a broad class of path transformations.

We empirically demonstrate the applicability of our approach by using HADES to synthesize bash scripts and XSLT programs for various tasks obtained from online forums. Our evaluation shows that HADES is practical, both in terms of running time and the input required from end-users.

2. Overview

In this section, 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. Specifically, Bob has a Music folder containing subdirectories such as Rock, Classical etc., and each of these folders in turn contains a collection of music files and subfolders (e.g., one subfolder for each band). Furthermore, suppose that Bob’s music files come in three different formats: mp3, ogg, and flac. However, since not every music player supports all of these different 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 also wants to convert all his flac files to mp3 and keep both the original as well as the converted files.

We now describe how Bob can use the HADES system for synthesizing a bash script that performs the exact transformation he wants. To use HADES, Bob starts by constructing the small input-output example shown in Figure 1. Specifically, the left hand side of Figure 1 corresponds to the input directory, and the right hand side shows the desired output directory. As an example, consider the file called `screenplay.flac` under the Jazz subfolder in the original directory. In the output directory, there are two files called `screenplay.flac` and `screenplay.mp3`. The latter one appears in the mp3 subdirectory under the Jazz folder, while `screenplay.flac` appears under the Jazz subfolder of the flac directory.

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 constructs a new set of input-output examples $\mathcal{E}$ where each example $e \in \mathcal{E}$ represents a path transformation. Specifically, for the directories shown in Figure 1, HADES generates the path transformation examples $\mathcal{E}$ shown in Figure 2. Each path transformation example $e \in \mathcal{E}$ consists of a pair of paths $(p_1, p_2)$ 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 sequence of pairs $(t, d)$ where $t$ is the label associated with a node in the HDT and $d$ is the corresponding data. In this particular application domain, labels corresponds to directory/file names, and data includes information about permissions, owner, file type etc. Observe that each path in the input HDT may appear in multiple examples of $\mathcal{E}$ due to duplication; for instance, input path $p_1$ in Figure 2 appears in both examples $\mathcal{E}_1$ and $\mathcal{E}_2$.

Once HADES constructs the path transformation examples $\mathcal{E}$, it then proceeds to synthesize a path transformation function $f$ such that $p' \in f(p)$ for every example $(p, p') \in \mathcal{E}$. Specifically, a path transformer is a function that takes an input path and computes a set of output paths. Note that path transformers return a set rather than a single path because we allow duplication as well as 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, and in the second phase, we perform classification by finding a predicate that differentiates each input path in one partition from other inputs paths in the remaining partitions. As mentioned in Section 1, we perform unification using SMT solving and classification using decision tree learning.

Going back to our example, HADES partitions examples $\mathcal{E}$ into two groups $\mathcal{P}_1, \mathcal{P}_2$, where $\mathcal{P}_1$ contains $\{\mathcal{E}_3, \mathcal{E}_4, \mathcal{E}_5\}$ and $\mathcal{P}_2$ contains $\{\mathcal{E}_2, \mathcal{E}_6\}$. For partition $\mathcal{P}_1$, we infer the unifier $\chi_1$:

$$
\text{concat} ( \text{map Id subpath}(x, 1, 1), \\
\text{map ExtOf subpath}(x, \text{size}(x), \text{size}(x)), \\
\text{map Id subpath}(x, 2, \text{size}(x)) )
$$

where 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, for partition $\mathcal{P}_2$, we infer the following unifier $\chi_2$, where FlacToMp3 is a function for converting flac files to mp3:
Figure 2. Path transformation examples constructed by HADES

Figure 3. Bash script synthesized by HADES

Next, we find a predicate for each partition by performing classification. Specifically, the input paths for partition $P_1$ are $\{p_1, p_2, p_3, p_4\}$, and the ones for $P_2$ are $\{p_1, p_3\}$. Since all input paths of $E$ are also in $P_1$, we infer the classifier $\phi_1 : \text{true}$ for $P_1$. On the other hand, for partition $P_2$, a simplest predicate that differentiates $\{p_1, p_2\}$ from $\{p_2, p_3\}$ is $\phi_2 : \text{ext = “flac”}$. Hence, the overall path transformer inferred by our algorithm is:

$$\lambda x. (x_1 ; \text{if( ext = “flac” ) then x2})$$

The final step of our algorithm is to synthesize a tree transformation using this inferred path transformer. For this purpose, HADES generates the (pseudo-) bash script shown in Figure 3. Effectively, the synthesized script iterates over every file $f$ in the input directory, applies the synthesized path transformer to $f$, and obtains a new set of output paths. Then, it creates the new output directory by calling a pre-defined function called makeDirectories which creates a new directory structure from the output paths.

Finally, going back to our motivating end-user scenario, Bob can now apply the script synthesized by HADES 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 semi-structured data such as file system directories and XML documents. We will also prove some important properties of HDTs that will be useful later.

Definition 1. (Hierarchical data tree) 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 $L : V \rightarrow \text{Id}$ assigns a label to each node $v \in V$.

The data store $D : V \rightarrow \text{Dat}$ maps each node $v \in V$ to the data associated with $v$.

An important point about hierarchical data trees is that the labeling function $L$ does not need to be one-to-one. That is, for nodes $v$ and $v'$, it is possible that $L(v) = L(v')$. With slight abuse of notation, we write $L(V)$ to mean the multi-set $\{\ell \mid v \in V \land L(v) = \ell\}$. Similarly, we write $L(E)$ to denote the multi-set:

$$\{(\ell, \ell') \mid (v, v') \in E \land L(v) = \ell \land L(v') = \ell'\}$$

Example 1. Each directory in a file system can be viewed as a hierarchical data tree where vertices correspond to files and directories, and an edge from $v$ to $v'$ indicates that $v$ is the parent directory of $v'$. The labeling function assigns, to each node $v$, the name of the file or directory associated with $v$. The data store $D$ assigns internal nodes (i.e., directories) to the metadata associated with that directory (e.g., permissions, creation date etc.). Similarly, $D$ assigns each leaf node (i.e., file) to its content and metadata.

Example 2. An XML file can also be viewed as a hierarchical data tree where nodes correspond to XML elements, and an edge from $v$ to $v'$ indicates that $v$ is nested directly inside element $v'$. The labeling function $L$ assigns a label $(s, i)$ to each element $v$, 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.

Notation. Given a hierarchical data tree $T = (V, E, L, D)$, we write $\text{root}(T)$ to indicate the unique root of the tree. For a given node $v$, we write $\text{parent}(v)$ to denote the unique node $v'$ such that $(v', v) \in E$. We define $\text{children}(v)$ as $\{v' \mid (v, v') \in E\}$. Finally, we write $\text{subtree}(T, v)$ to denote the subtree of $T$ rooted at node $v$, and we write $\text{isLeaf}(T, v)$ to indicate that $v$ is a leaf node in $T$.

Definition 2. (Well-formedness) We say that a hierarchical data tree $T = (V, E, L, D)$ is well-formed iff no two sibling vertices have the same label, i.e.:

$$\forall v, v'. (\text{parent}(v) = \text{parent}(v') \land v \neq v') \Rightarrow L(v) \neq L(v')$$

In later sections of this paper, we will assume that all hierarchical data trees are well-formed, and we will use the term “tree” to mean a well-formed hierarchical data tree. We note that our 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).
For every \( i \in [1, k] \), there exists an edge \((v, v') \in E\) such that \( L(v) = \ell_i, L(v') = \ell_{i+1}, D(v) = d_i, \) 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 \( \ell_i \) and \( d_i \) respectively. The set of paths in a tree \( T \) is denoted by \( \text{paths}(T) \), and we will write \( \text{pathTo}(T, v) \) to denote a path starting at the root of \( T \) and ending in \( v \). Note that \( \text{pathTo}(T, v) \) ends in \((\ell, d)\) such that \( L(v) = \ell \) and \( D(v) = d \).

**Proposition 1.** Let \( p \) be a path in a well-formed hierarchical data tree \( T = (V, E, L, D) \). There exists a unique \( v \in V \) such that \( \text{pathTo}(T, v) = p \).

Observe that this proposition follows immediately from the definition of well-formedness. We now define what it means for two HDTs to be equivalent:

**Definition 4. (Equivalence of HDTs)** 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 \( v_1' \in \text{children}(v_1) \) and \( v_2' \in \text{children}(v_2) \) such that \( L_1(v_1') = L_2(v_2') \), then \( \text{subtree}(T_1, v_1') = \text{subtree}(T_2, v_2') \).

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

**Theorem 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 is important for our approach because the programs we synthesize construct the output tree from a set of paths. In particular, this theorem states that a given set of paths uniquely defines a well-formed hierarchical data tree. In contrast, if we lift the well-formedness assumption, then a set of paths does not define a unique hierarchical data tree. This point is illustrated by the following example.

**Example 3.** Consider the two trees shown in Figure 4, where the letters on each node indicate their label, and assume that all nodes store the same data \( d \). For both trees, \( \text{paths}(T) \) is the set:

\[
\{(A, d), (B, d), (C, d), [(A, d), (B, d), (D, d)]\}
\]

However, observe that the left tree \( T_1 \) violates the well-formedness assumption because \( A \) has two children with label \( B \). As this example illustrates, a set of paths does not uniquely define an HDT if we remove the well-formedness assumption. Fortunately, well-formedness guarantees that we can always reconstruct an HDT from its constituent paths.

---

2. Proofs of all non-trivial statements are given in the appendix.

---

**4. Synthesizing Trees from Paths**

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

**Overview.** As summarized in Figure 5, the high-level structure of our synthesis algorithm consists of four steps. First, given a set of input-output HDTs \( \mathcal{E} \), we start by verifying that every example in \( \mathcal{E} \) conforms to a certain unambiguity restriction required by our algorithm; this is done by calling a function called \( \text{CHECKEX} \) at line 4. Next, we \( \text{furcate} \) the input-output trees \( \mathcal{E} \) into a set \( \mathcal{E}' \) of path transformation examples. Specifically, each example \( e \in \mathcal{E}' \) maps a path \( p \) in some input tree \( T \) to a “corresponding” path \( p' \) in \( T' \) for some \((T, T') \in \mathcal{E}\). Now, given the furcated examples \( \mathcal{E}' \), we invoke a function called \( \text{INFERP} \) to learn an appropriate path transformer \( f \) such that \( p' \in f(p) \) for every \( (p, p') \in \mathcal{E}' \). Finally, \( \text{CODE} \) 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 each of these steps in more detail, leaving the \( \text{INFERP} \) 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 \( \Pi \) 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 \( \mathcal{E} \), let us define \( \mathcal{E}.in \) to be the input trees in \( \mathcal{E} \) (i.e., \( \mathcal{E}.in = \{T \mid (T, T') \in \mathcal{E}\} \)). Similarly, let \( \mathcal{E}.out \) be the set of output trees in \( \mathcal{E} \). Our synthesis algorithm expects the user-provided examples \( \mathcal{E} \) to obey a certain semantic unambiguity criterion, defined as follows:

**Unambiguity:** For every \( p' \in \text{paths}(\mathcal{E}.out) \), there exists a unique \( p \) such that \( p \sim p' \) where \( p \in \text{paths}(\mathcal{E}.in) \) and \( (p, p') \in \text{paths}(T) \times \text{paths}(T') \) for some \((T, T') \in \mathcal{E}\).

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. This unambiguity criterion is enforced by the function \( \text{CHECKEX} \) used at line 4 of the \( \text{SYNTHESIZE} \) algorithm.

The correspondence relation \( \sim \) can be defined in many natural ways. Specifically, the \( \text{AD} \) system allows the user to mark paths in the input-output examples as corresponding. However, \( \text{AD} \) also comes with a default definition of \( \sim \) that is adequate in many practical settings. Let \( p = [(\ell_1, d_1), \ldots, (\ell_k, d_k)] \) and \( p' = [(\ell'_1, d'_1), \ldots, (\ell'_{k'}, d'_{k'})] \) be two different paths. By default, we define \( p \sim p' \) if \( \ell_k = \ell'_{k'} \) (i.e., \( p \) and \( p' \) end in leaves with the same label). When the default definition of \( \sim \) is used, \( \text{AD} \) uses a refinement of the unambiguity criterion that can be checked syntactically and efficiently.

**R1.** Let \((T, T') \in \mathcal{E}\). For every leaf \( v' \) of \( T' \), there exists a leaf \( v \) of \( T \) such that \( L(v') = L(v) \).
R2. If \(v_1, v_2\) are leaves in \(E.in\) where \(v_1 \neq v_2\), then \(L(v_1) \neq L(v_2)\).

It is easy to see that syntactic requirements R1 and R2 are sufficient to guarantee unambiguity. In particular, requirement R1 ensures existence, while R2 guarantees uniqueness.

**Furcation.** Given an unambiguous set of examples \(E\), our algorithm furcates them into a set of path transformation examples \(E'.\) Specifically, a pair of paths \((p, p') \in E'\) iff:

\[ p \in \text{paths}(T), p' \in \text{paths}(T'), \text{ and } p \sim p' \]

for some \((T, T') \in E\). If some input path \(p \in \text{paths}(T)\) does not have a corresponding output path \(p' \in \text{paths}(T')\), then \(E'\) also contains \((p, \perp)\).

Note that \(E'\) may contain multiple examples that have path \(p\) as an input — i.e., it is possible that \((p, p')\) and \((p, p'')\) are both in \(E'\). For instance, this can happen if some leaf in the input tree has been duplicated in the corresponding output tree. However, due to the unambiguity requirement, it is not possible that there are multiple examples in \(E'\) that have the same output path. That is, if \((p, p) \in E'\), then there does not exist another example \((p', p') \in E'\) where \(p \neq p'\).

Given a set of path transformation examples \(E'\), we will write \(\text{inputs}(E')\) (resp. \(\text{outputs}(E')\)) to denote the input paths (resp. output paths) in \(E'\). That is, \(p \in \text{inputs}(E')\) iff there exists an example \((p, \perp) \in E'\) and \(p' \in \text{outputs}(E')\) iff there exists some \((\perp, p') \in E'\).

**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 transformation function \(f\) from the furcated examples \(E'\). A path transformer is a function that takes as input a path and returns a set of output paths. Specifically, any path transformer \(f\) returned by \(\text{inferPathTrans}\) at line 7 of the \(\text{synthesize}\) procedure must satisfy the following requirement:

\[ \forall p \in \text{inputs}(E'), \exists (p' \in f(p)) \wedge (p, p') \in E' \]

Note that when an input path \(p\) does not have a corresponding output path \(p'\), this requirement implies that \(f(p) = \{\perp\}\). In the remainder of this section, we will assume the existence of an oracle that can synthesize such path transformers, and we will revisit this topic in Section 5.

**Code generation using splicing.** 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, \(\text{splice}(S)\) yields a well-formed tree \(T\) such that \(\text{paths}(T) = S\). Recall from Theorem 1 that the result of splicing must be unique.

The program generated by \(\text{codeGen}\) is shown in Figure 7. The procedure \(\text{transform}\) synthesized by our algorithm takes as input any well-formed tree \(T\) and returns the result of applying the desired transformation to \(T\). In particular, given input tree \(T\),

1: procedure \(\text{transform}(\text{Tree} T)\)
2: \(\text{Input:}\) Well-formed tree \(T\)
3: \(\text{Output:}\) Result of desired transformation
4: \(S := \text{paths}(T);\)
5: \(S' := \emptyset;\)
6: for all \(p \in S\) do
7: \(\Pi := f(p);\)
8: if \(\Pi \notin \{\perp\}\) then
9: \(\text{insert}(S', \Pi);\)
10: \(T' := \text{splice}(S');\)
11: return \(T'\);
Path transformer $\tau := \lambda x. \{ \phi_1 \rightarrow \chi_1 \oplus \ldots \oplus \phi_n \rightarrow \chi_n \}$

Path term $\chi := \text{concat}(\{ \tau_1(x), \ldots, \tau_n(x) \})$

Segment transformer $\tau := \lambda x. \text{map } F \text{ subpath}(x, t_1, t_2)$

Index term $t := b \cdot \text{size}(x) + c$ where $b$ is either 0 or 1, $c$ is an integer, and $\text{size}(x)$ denotes the number of elements in path $x$.

Mapper $F := \lambda x. \text{int} \cdot \text{int} \cdot f_i(x)$ where $\phi_i$ is drawn from a family of pre-defined predicate templates (e.g., for checking file type).

Overview. We now give a high-level overview of our algorithm for learning path transformers. As illustrated in Figure 9, our algorithm consists of three key components, namely partitioning, unification, and classification. The goal of partitioning is to determine the smallest arity path transformer $\pi$ that can be used to describe the desired transformation and divide the examples $E$ into groups of unifiable subsets. We say that a set of examples $E^*$ is unifiable if outputs($E^*$) can be represented using the same path term $\chi^*$ and we refer to $\chi^*$ as the unifier for $E^*$. Our algorithm represents each partition $P_i$ as a triple $(E_i, \chi_i, \phi_i)$ where $E_i$ is a unifiable set of examples, $\chi_i$ is their unifier, and $\phi_i$ is a predicate distinguishing $E_i$ from the other examples.

The partitioning component of our algorithm is based on an enumerative search that tries different hypotheses. In our algorithm, a hypothesis corresponds to a fixed-size partitioning of examples $E$ into $k$ disjoint groups $E_1, \ldots, E_k$. For each hypothesis, we invoke the unification module to check whether each of $E_1, \ldots, E_k$ is unifiable, and, if so, what their unifier is. Hence, the unification component can be viewed as an oracle for confirming a given hypothesis. If the unification procedure confirms the feasibility of a specific hypothesis, we then invoke the classification module to find a predicate that differentiates each partition from all other partitions.

Since our algorithm repeatedly invokes the unification component to confirm or refute a specific 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 $\chi_i$ associated with a set of examples $E_i$ 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 $E_1, \ldots, E_k$, the goal of classification is to infer a predicate $\phi_i$ for each $E_i$ such that $\phi_i$ evaluates to true for each $p \in \text{inputs}(E_i)$ and evaluates to false for each $p' \in \text{inputs}(E_i) - \text{inputs}(E_i)$. 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 10 presents pseudo-code for the InferPathTrans procedure based on this discussion. The algorithm consists of two phases: In the first phase, we partition examples $E$ into a set $\Phi$ of unifiable groups, and, in the second phase, we infer classifiers for each partition. Specifically, lines 5-7 try to partition $E$ into $i$ disjoint groups by invoking the PARTITION procedure. Hence, after this loop, set $\Phi$ is a partitioning of $E$ into the minimum number of unifiable groups. In the second phase (lines 9-12), we then infer a classifier $\phi_i$ for each partition $P_i$ in $\Phi$. Finally, we use a procedure called PtCodeGen to generate a path transformer $\pi$ from the partitions. In what follows, we describe partitioning, unification, and classification in more detail.

5.1 Partitioning

Figure 11 shows the partitioning algorithm used in InferPathTrans. The recursive PARTITION procedure takes as input a set of examples $E_i$ that are part of the same partition, the remaining examples $E_2$, 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 $E_1 \cup E_2$, and, if this is not possible, we return failure (i.e., $\emptyset$).

In the recursive case (lines 9-17), we try to grow the current partition $E_1$ by adding one or more of the remaining examples from $E_2$. The algorithm always maintains the invariant that elements in $E_1$ are unifiable. Hence, we try to add an element $e \in E_2$ to $E_1$ (line 10), and if the resulting set is not unifiable, we give up and try a different element (lines 11-12). Since we know that $E_1 \cup \{ e \}$ is unifiable, we now see if it is possible to partition the remaining examples $E_2 - \{ e \}$ into $k - 1$ unifiable sets: this corresponds to the recursive call at line 13. If this is indeed possible (i.e., the recursive call does not return $\emptyset$), we have found a way to partition $E_1 \cup E_2$ into $k$ different partitions and return success (line 15).

Now, if the remaining examples $E_2$ cannot be partitioned into $k - 1$ unifiable sets, we try to shrink $E_2$ by growing $E_1$. Hence, the recursive call at line 16 looks for a partitioning of examples where one of the partitions contains at least $E_1 \cup \{ e \}$. 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 \( \text{PARTITION}(\emptyset, \mathcal{E}, k) \) effectively explores all possible ways to partition examples \( \mathcal{E} \) into \( k \) unifiable subsets. However, since many subsets of \( \mathcal{E} \) are typically not unifiable, the algorithm does not exhibit its worst-case \( O(k^n) \) behavior in practice.

### 5.2 Unification

We now describe the UNIFY procedure for determining if a set of examples \( \mathcal{E} \) has a unifier. Since the unification algorithm is invoked many times during partitioning, we want to ensure that UNIFY is efficient in practice. To achieve this goal, we formulate unification 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" \( D \) to map \( e \) to a numerical value that is outside of the index range of the input paths. More formally, we define example summarization as follows:

**Definition 5. (Example summarization)** Let \( \mathcal{E} \) be a set of examples where \( \mathcal{E} \) denotes the set of labels used in \( \mathcal{E} \), and let \( \mathcal{F} \) be the set of pre-defined functions allowed in the path transformer. Let \( D: (\mathcal{E} \cup \mathcal{F}) \to \{i | i \in \mathbb{Z} \wedge i > m\} \) 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 \((n, \sigma)\) where \( n \) is the length of path \( p_1 \) and \( \sigma \) is a sequence such that:

\[
\sigma_i = \begin{cases} (j, p_1[j], d \to p_2[i], d) & \text{if } \exists j, p_1[j], \ell = p_1[j].\ell \\ (D(p_2[i], \ell), \ l \to p_2[i], d) & \text{otherwise} \end{cases}
\]

We illustrate summarization using a few examples:

**Example 5.** Consider the example \((p_1, p_2)\) such that \( p_1 \) is the path \([A, r), (B, r), (C, r)]\), and \( p_2 = [(C, w), (A, r), (B, r)]\) where \( A, B, C \) are labels corresponding to directory names, and \( r \) and \( w \) indicate read/write permission. The summarized example is \((3, \sigma)\) where \( \sigma = [(3, \rightharpoonup w), (1, \rightharpoonup r), (2, \rightharpoonup r)]\). The first element in \( \sigma \) is \((3, \rightharpoonup w)\) because \( C \), which is the first element in the output path, is at index 3 in the input path, and its corresponding data is mapped from \( r \) to \( w \). Note that the sequence \([3, 1, 2]\) describes how elements in the input have been permuted to obtain the output.

**Example 6.** Now, consider the same input path \( p_1 \) from Example 5, but a different output path \( p_2' = [(A, \rightharpoonup), (B, pdf)]\), and the output path \([A, \rightharpoonup), (pdf, \rightharpoonup), (B, pdf)]\). Also, suppose that \( D(\text{New}) = 1000 \) (i.e., our "dictionary" assigns value 1000 to foreign element \( \text{New} \)). In this case, the summarized example is \((3, \sigma')\) where \( \sigma' = [(1, \rightharpoonup r), (2, \rightharpoonup r), (1000, \rightharpoonup r)]\). Note that \( \text{New} \) is mapped to 1000 using case 2 of Definition 5.

**Example 7.** Consider the input path \([A, \perp), (B, pdf)]\) and the output path \([A, \perp), (pdf, \perp), (B, pdf)]\). Also, suppose that we have a built-in function called \( \text{ExtOf} \) that allows us to retrieve the type (extension) of a given file. In this case, the summarized example is \((2, \sigma)\) where \( \sigma \) is given by:

\[
[(1, \perp \rightharpoonup \perp), (D(\text{ExtOf}) + 2, \perp \rightharpoonup \perp), (2, pdf \rightharpoonup pdf)]
\]

Note that label \( pdf \) in the output list is mapped to \( D(\text{ExtOf}) + 2 \) because it corresponds to the extension for element at index 2 in the input list (case 2 of Definition 5). In general, adding index \( j \) to \( D(f^*) \) in case 2 of Definition 5 has two advantages: First, it allows us to encode the path element to which \( f^* \) was applied. Second, if the same function is applied to contiguous elements in the input path, then we will obtain a contiguous range of numbers, which we can then coalesce into a loop construct.\(^\text{5}\)

---

\(^5\)For reasons that will become clear later, we also require that \( D \) satisfies \( \forall t, t' \neq t' \Rightarrow |D(t) - D(t')| \geq m \). While this is not necessary for the correctness, it makes our unification algorithm more efficient.

\(^6\)In the general case, note that the \( j \) index used in Definition 5 may not be unique, in which case our algorithm considers all possible summarizations. However, we have never observed this to be the case in practice, and we make the uniqueness assumption to simplify our presentation.
Given a summarized example \( e = (n, \sigma) \), we say that \( e^* \) is a consolidated form of \( e \) if it is of the form \((n, \sigma^*)\) where \( \sigma^* = \{(b_1, e_1, M_1), \ldots, (b_k, e_k, M_k)\} \) and:

- \( \text{indices}(e) = [b_1, \ldots, b_k, e_1, \ldots, e_k] \)
- For all \( i, [b_i, b_{i+1}, \ldots, e_i] \) is a contiguous sublist of \( \text{indices}(e) \)
- \( M_k = \bigcup_m \{m_j \mid b_k \leq j \leq e_k \land \sigma = (i_j, m_j)\} \)

Intuitively, consolidated form can coalesce consecutive indices in a summarized example. Note that the consolidation of a summarized 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 consolidated forms:

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

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

Note that, in \( e_2^* \), we have coalesced the two contiguous elements from the input path into the same element \( (1, 2, \{r \mapsto r\}) \).

Given a consolidated example \( e^* = (n, \sigma^*) \) where \( \sigma^* = \{(b_1, e_1, M_1), \ldots, (b_k, e_k, M_k)\} \), we define \( \text{len}(e^*) \) to be \( n \) and \( \text{segments}(e^*) \) to be \( k \). We also write \( \text{begin}(e^*, \sigma) \) to indicate \( b_j \), \( \text{end}(e^*, \sigma) \) for \( e_j \), and \( \text{data}(e^*, \sigma) \) for \( M_j \). Note that \( e^* \) can be viewed as a concatenation of concrete path segments of the form \((c, c', M)\) 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 \((t, t', F)\)\(^7\) in the language from Figure 8, and each concrete path segment \((c, c', M)\) in a consolidated example. Specifically, observe that a concrete path segment can be viewed as a concrete instantiation of a segment transformer \( (b + \text{size}(x) + c, b' + \text{size}(x) + c', F) \) where each of the terms \( b, b', c, c', F \) 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 consolidated examples to solve for the unknown terms in segment transformers using an SMT solver.

We are now ready to explain the details of our unification algorithm, which is presented in Figure 12. Given a set of examples \( \mathcal{E} \), the UNIFY algorithm starts by computing the summarized examples \( \mathcal{E}' \) at line 5 and generates all possible consolidated forms for each \( e'_i \in \mathcal{E}' \) (line 6). Now, since we don’t know which consolidated form is the “right” one, we need to consider all possible combinations of consolidated forms of the examples. Hence, set \( \Lambda \) obtained at line 6 corresponds to the Cartesian product of the consolidated form of examples \( \mathcal{E}' \).

Next, the algorithm enumerates all possible candidate unifiers \( \chi \) of increasing arity, where arity refers to the number of path segments being concatenated. Based on the grammar of our language (recall Figure 8), we know that a path term \( \chi \) of arity \( k \) has the form \( \text{concat}(\tau_1(x), \ldots, \tau_k(x)) \) and each \( \tau_i \) is a segment transformer described by the following template:

\[
(b_i \cdot \text{size}(x) + c_i, b'_i \cdot \text{size}(x) + c'_i, F_i)
\]

Hence, the hypothesis \( \chi \) at line 10 of the algorithm is a templateized unifier whose unknown coefficients will later be inferred if the examples can indeed be unified using an arity-\(k\) path term.

---

\(^7\)Recall that \((t, t', F)\) is an abbreviation for \(\lambda x.\ \text{map} F\ \text{subpath}(x, t, t')\).
Examples $E^*$ and look for a different combination of consolidated examples satisfying $\chi$.

If, however, $\phi$ is satisifiable, we have found an instantiation of the unknown coefficients, which is given by the satisfying assignment $\sigma$ obtained at line 20. Now, the only question that remains to be answered is whether we can also find an instantiation of the unknown functions $F_1, \ldots, F_k$ used in $\chi$. 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 8 only allows a finite set of pre-defined data transformers $f_i$ and predicates $\phi_i$, UNIFYMAPPERS enumerates in increasing order of complexity– all possible functions belonging to the grammar of mapper functions in Figure 8.

**Example 9.** For the motivating example from Section 2, our unification algorithm takes the following steps to determine the unifier $\chi_1$ for partition $\mathcal{P}_1$: First, it generates the summarized examples shown in Figure 13. Next, it generates all possible consolidated examples $\sum \mathcal{E}$ shown in Figure 14. We generate the formula $\phi$ shown in Figure 15 and get a satisfying assignment, which results in the following instantiation of $\chi$:

$\{1, 1, F_1\}, \langle D\langle \text{ExtOf}\rangle + \text{size}(x), D\langle \text{ExtOf}\rangle + \text{size}(x), F_2\\rangle, \langle 2, \text{size}(x), F_3\rangle$.

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 $\text{extOf}$ from the segment transformer coefficients. As a result, we obtain the unifier $\chi_1$ presented in Section 2.

**Discussion.** We now briefly discuss why we use constraint solving instead of enumerative search for finding the coefficients used in each hypothesis. Recall that each index term in our language is of the form $b \cdot \text{size}(x) + c$ where $b$ is either 0 or 1 and $c$ is an integer, where $|c|$ must be in the range $[0, \text{size}(x)]$. Hence, for a hypothesis of arity $k$, the size of the search space that we would need to explore is $(2 \cdot 2 \cdot \text{maxSize}(\text{inputs}(\mathcal{E})))^k$. Since solving linear equalities using an SMT solver is, in practice, much more efficient than searching through spaces of this size, performing unification using constraint solving makes our approach much more practical.

### 5.3 Classification

We now turn to the last missing piece of our algorithm for learning path transformations. Recall that the second phase of our INFERPATHTRANS procedure needs to find a predicate differentiating a given partition from all other partitions. Specifically, given examples $\mathcal{E}$ and a partition $\mathcal{P}_i$ with examples $\mathcal{E}_i \subseteq \mathcal{E}$ and a unifier $\chi_i$, the goal of classification is to find a predicate $\phi_i$ such that:

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

In other words, we want to find a predicate $\phi_i$ that evaluates to true for all inputs in $\mathcal{E}_i$ and to false for all other inputs in $\mathcal{E}$. As a practical matter, we also want each $\phi_i$ to be as simple as possible because simpler predicates tend to generalize better and lead to more general programs (i.e., the principle of Occam’s razor).

A key insight underlying our algorithm is that finding a predicate $\phi_i$ with these properties is precisely the familiar classification problem in machine learning. Hence, to find such a predicate $\phi_i$, we first extract the relevant features for each path and then use the ID3 decision tree learning algorithm.

**Feature extraction.** To use decision tree learning for classification, we need to represent each input path using a finite set of discrete features. In our \textsc{Hades} system, the set of features used to represent a given path is domain-specific and is defined separately for each application domain (e.g., UNIX directories or XML documents). As an example, some of the features we use for the file system domain include the following:

- The file type (e.g., mp3, pdf) associated with the leaf
- A boolean value indicating whether the path contains a certain file or directory
- The permissions associated with the leaf
- A boolean value indicating whether the path contains a directory owned by “root”

In addition to these domain-specific attributes, one additional feature we always include, regardless of the application domain, is the label associated with the leaf. In particular, since we require that each leaf in the input tree has a unique label, the inclusion of leaf labels as a feature guarantees that a classifier always exists. Given a path $p$, we write $\alpha(p)$ to denote the feature vector for $p$, and we write $\alpha_i(p)$ to denote the value of feature $f$ for path $p$.

**Decision tree learning.** We now explain how we use decision tree learning to infer a predicate distinguishing paths $\Pi_1$ from a different set of paths $\Pi_2$. Here, $\Pi_1$ corresponds to $\text{inputs}(\mathcal{E}_i)$ where $\mathcal{E}_i$ is the set of examples for $\mathcal{P}_i$, and $\Pi_2$ corresponds to $\text{inputs}(\mathcal{E}) - \text{inputs}(\mathcal{E}_i)$.

Given these sets $\Pi_1$ and $\Pi_2$ and a set of features $F$, we use the ID3 algorithm [29] to construct a decision tree $\mathcal{T}_D$ with the following properties:

- Each leaf in $\mathcal{T}_D$ is labeled as $\Pi_1$ or $\Pi_2$.
- Each internal node of $\mathcal{T}_D$ is labeled with a feature $f \in F$.
- Each directed edge $(f, f', \ell)$ from node $f$ to $f'$ is annotated with a label $\ell$ that corresponds to a possible value of feature $f$.
- Let $(f_1, f_2, \ell_1), \ldots, (f_n, \Pi_1, \ell_n)$ be a root-to-leaf path in $\mathcal{T}_D$. Then, for every $p \in \Pi_1 \cup \Pi_2$, the following condition holds:

$$\bigwedge_{i=1}^{n} \alpha_i(p) = \ell_i \Leftrightarrow p \in \Pi_i$$

We note that the ID3 algorithm also attempts to minimize the size of the constructed decision tree. However, since it uses a greedy approach based on information gain, it only guarantees local–but not global–optimality.

Once we construct such a decision tree $\mathcal{T}_D$ with the aforementioned properties, identifying a predicate $\phi$ differentiating $\Pi_1$ from $\Pi_2$ is very simple. Specifically, given a root-to-leaf path $\pi = ((f_1, f_2, \ell_1), \ldots, (f_n, \Pi_1, \ell_n))$ ending in a leaf with label $\Pi_1$, let $\varphi(\pi)$ denote the formula:

$$\varphi(\pi) : \bigwedge_{i=1}^{n} f_i = \ell_i$$

Assuming $\Pi_1$ corresponds to $\text{inputs}(\mathcal{E}_i)$ for some $\mathcal{P}_i$ with examples $\mathcal{E}_i$ and $\pi_2$ is $\text{inputs}(\mathcal{E}) - \text{inputs}(\mathcal{E}_i)$, the following DNF formula $\phi$ gives us a classifier for partition $\mathcal{P}_i$:

$$\phi_i : \bigvee_{\pi \in \text{path}(\mathcal{T}_D, \Pi_1)} \varphi(\pi)$$

In particular, observe that $\phi_i$ evaluates to true for any path $p \in \Pi_1$ because at least one of the disjuncts of $\phi_i$ will evaluate to true. In contrast, $\phi_i$ evaluates to false for any $p' \in \Pi_2$ because all disjuncts evaluate to false when $\phi_i$ is obtained from a valid decision tree $\mathcal{T}_D$. 
Example 10. Recall the motivating example from Section 2 where inputs(ℰ) = {p₁, p₂, p₃, p₄}. Let us now consider how we find a predicate for partition ℙ₁. Here, we have Π₁ = inputs(ℰ₂) = {p₁, p₃} and Π₂ = inputs(ℰ) − inputs(ℰ₂) = {p₂, p₄}. After running the ID3 algorithm, we obtain the following decision tree:

![Decision Tree Diagram]

Hence, we extract the classifier φ₂ : ExtOf(x) = flac.

5.4 Soundness of Path Transformation Synthesis

We conclude this section by stating a key property of the INFERPATHTRANS algorithm.

Theorem 4. Let f be a function synthesized by INFERPATHTRANS using path transformation examples ℰ. Then,

\[ \forall p \in \text{inputs}(ℰ). \ (p' \in f(p) \iff (p, p') \in ℰ) \]

This theorem is very important for the soundness of our algorithm for synthesizing tree transformations (recall Section 4). In particular, without this property, we cannot guarantee that, for an input-output example (T₁, T₂), applying TRANSFORM to T₁ will actually yield T₂.

6. Implementation

We have implemented the proposed algorithm for synthesizing HDT transformations in a tool called HADES, which currently consists of ~ 9,500 lines of C++ code (including our implementation of the ID3 decision tree learning algorithm). The only external tool used by HADES is the Z3 SMT solver [10].

As mentioned in Section 1, the core of HADES is the domain-agnostic synthesis backend for learning HDT transformations from examples. The synthesis back-end accepts input-output examples in the form of hierarchical data trees and emits path transformation functions in an intermediate language (recall Figure 8).

In order to allow our system to be easily extensible to new domains, HADES also provides an interface for domain-specific plug-ins (front-ends). Our current implementation incorporates two such plug-ins, one for XML transformations in the XSLT scripting language and another one for file system transformations as bash scripts. Our XML front-end uses the tinyxml2 library for parsing XML documents.

HADES can be extended to new domains by implementing plug-ins that implement the following functionality:

1. Represent input-output examples as HDTs
2. Use the synthesized path transformer to emit tree transformation code in the target language
3. Specify the domain-specific functions that the path transformer is allowed to use — note that such domain-specific functions correspond to the pre-defined fᵢ’s from Figure 8
4. Define a feature extraction function as well as any domain-specific features to be used for decision tree learning (recall Section 5.3)

Assuming a plug-in can provide this functionality for domain D, HADES can immediately be used for synthesizing D-specific hierarchical data transformations.

7. Evaluation

We evaluated HADES by using it to synthesize a collection of 36 real-world data transformations in the file system and XML domains. Our examples mostly come from on-line forums such as Stackoverflow and bashscript.org. In addition, some of the examples in the file system domain were suggested to us by teaching assistants who needed to write bash scripts for various tasks in undergraduate computer science courses at our institution.

In addition to assessing whether HADES can successfully synthesize these benchmarks, we also want to evaluate the following:

• **Performance:** How long does HADES take to synthesize the desired transformation?
• **Complexity:** How complex are the path transformations synthesized by HADES, both in terms of lines of code as well as other metrics (e.g., number of loops)?
• **Usability:** How hard is it for users to work with HADES? How many input-output examples do users need to provide before HADES learns the desired transformation?

In order to answer these questions, we performed a user study in which we asked a group of students—some computer science majors, others from different disciplines—to work with HADES and synthesize a script that performs a specific task chosen from Figure 16. We emphasize that none of the students in our study had prior knowledge of HADES; furthermore, no participant in our study is familiar with program synthesis research.
### Benchmarks

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
<th>Script</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>Branches</td>
<td>Segments</td>
<td>LOC</td>
</tr>
<tr>
<td>F1</td>
<td>Categorize .csv files based on their group</td>
<td>0.03</td>
<td>1</td>
</tr>
<tr>
<td>F2</td>
<td>Make all script files executable</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>F3</td>
<td>Copy all text and bash files to directory temp</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>F4</td>
<td>Append last 3 directory names to the file name and delete those directories</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>F5</td>
<td>Put files in directories based on modification year/month/day</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>F6</td>
<td>Copy files without extension into the &quot;Notextension&quot; directory</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>F7</td>
<td>Archive each directory to a tarball with modification month in its name</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>F8</td>
<td>Make files in &quot;DoNotModify&quot; directory read-only</td>
<td>0.12</td>
<td>2</td>
</tr>
<tr>
<td>F9</td>
<td>Convert mp3, wma, and .ma4 files to .ogg</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>F10</td>
<td>Change group of text files in &quot;Public&quot; directory to &quot;everyone&quot;</td>
<td>0.06</td>
<td>2</td>
</tr>
<tr>
<td>F11</td>
<td>Change directory structure from contest/Sub/uid/pid/sid/to contest/Sub/pid/sid-uid/</td>
<td>0.03</td>
<td>1</td>
</tr>
<tr>
<td>F12</td>
<td>Convert .zip archives to tarballs</td>
<td>0.08</td>
<td>2</td>
</tr>
<tr>
<td>F13</td>
<td>Organize all files based on their extensions</td>
<td>1.85</td>
<td>4</td>
</tr>
<tr>
<td>F14</td>
<td>Append modification date to the file name</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>F15</td>
<td>Convert put files to swl files</td>
<td>0.03</td>
<td>2</td>
</tr>
<tr>
<td>F16</td>
<td>Delete files which are not modified last month</td>
<td>0.03</td>
<td>2</td>
</tr>
<tr>
<td>F17</td>
<td>Convert Video files to audio files and put them in &quot;Audio&quot; directory</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>F18</td>
<td>Convert xml files ≥1kB to text files</td>
<td>0.09</td>
<td>2</td>
</tr>
<tr>
<td>F19</td>
<td>In directory &quot;game&quot;: append &quot;largest&quot; to name of largest file, append &quot;big&quot; to name of xml files ≥1kB</td>
<td>17.94</td>
<td>3</td>
</tr>
<tr>
<td>F20</td>
<td>Extract tarballs to a directory with name combined of file and parent directory name</td>
<td>0.36</td>
<td>2</td>
</tr>
<tr>
<td>F21</td>
<td>Append parent directory name to each .c file and copy them in &quot;MOSS&quot; directory</td>
<td>0.04</td>
<td>2</td>
</tr>
<tr>
<td>F22</td>
<td>Keep all files older than 3 days</td>
<td>0.39</td>
<td>2</td>
</tr>
<tr>
<td>F23</td>
<td>Copy each file to the directory created with its file name</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>F24</td>
<td>Archive directories which are not older than a week</td>
<td>0.11</td>
<td>2</td>
</tr>
<tr>
<td>X1</td>
<td>Add style=&quot;bold&quot; attribute to parent element of each text</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>X2</td>
<td>Merge elements with attribute ‘status’ and put that attribute in their children</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>X3</td>
<td>Remove all attributes</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>X4</td>
<td>Change the root element of xml</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>X5</td>
<td>Remove 3rd element and put all nested elements inside it under its parent</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>X6</td>
<td>Create a table which maps each text to its parent element tag</td>
<td>0.09</td>
<td>2</td>
</tr>
<tr>
<td>X7</td>
<td>Remove all texts in element ‘done’ and put all remaining texts to ‘todo’ elements</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>X8</td>
<td>Generate HTML drop down list from a XML list storing the data</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>X9</td>
<td>Rename a set of element tags to standard HTML tags</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>X10</td>
<td>Move attribute ‘class’ to the first element and categorize based on ‘class’</td>
<td>0.18</td>
<td>2</td>
</tr>
<tr>
<td>X11</td>
<td>Categorize elements based on ‘tag’ attribute and put each class under an element with valid HTML tag</td>
<td>5.55</td>
<td>3</td>
</tr>
<tr>
<td>X12</td>
<td>Delete all elements with tag &quot;p&quot;</td>
<td>0.05</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 16.** File System and XML Benchmarks

### Setup
In our user study, we gave each participant six specific benchmarks to synthesize. In particular, we asked the participants to come up with a set of input-output examples until HADES synthesized the desired transformation. Since many of the participants are not familiar with bash or XSLT scripts, we also provided them with test cases. In particular, we asked the participants to verify the correctness of the synthesized script by running it on the provided tests and confirming that it produces the desired result. The participants were asked to provide HADES with a new set of examples — distinct from those in our test suite — if the synthesized script did not produce the desired output.

### Summary of results
Figure 16 summarizes the results of our evaluation for the file system and XML domains. The column labeled “Description” provides a brief summary of each benchmark, and “Time” reports synthesis time in seconds. The column labeled “Script” gives some statistics about the synthesized script, and the column labeled “User” provides some important data related to user interaction. We now describe each of these aspects in more detail.

### Performance
To evaluate performance, we utilized the examples provided by non-expert users from our user study. 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.

The column labeled “Time” in Figure 16 reports the total synthesis time in seconds. This time includes all aspects of synthesis, including conversion of examples to HDTs and emission of real 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. Observe that no single benchmark takes more than 18 seconds to synthesize. We believe these results demonstrate that HADES is practical and can be used in a real-world setting.

---

8 When there were multiple rounds of interaction with the user, we used the examples from the last round.
**Complexity.** The column labeled “Script” in Figure 16 reports various statistics about the script synthesized by HADES. In particular, the column labeled “branches” reports the number of top-level branches in the synthesized program (i.e., the arity of the path transformer). The column labeled “Segments” reports the number of loops (i.e., number of segment transformers). Finally, the column labeled “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; the statistics here only include the path transformation portion (recall that HADES also emits code for furcation and splicing). 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 section of Figure 16 reports various statistics from the user study. Specifically, the column labeled “Iteration” reports the number of rounds of tool-user interaction until HADES was able to synthesize the desired script (i.e., pass all test cases). The column labeled “Examples” summarizes the number of furcated examples in the input-output trees provided by the user. Finally, the column labeled “Depth” gives the maximum depth of the input-output trees. We remark that the number of examples reported here is not a lower bound on the number of examples that must be provided by the user. In particular, some users choose to provide more examples than necessary to increase their confidence or reduce the number of back-and-forth interactions with the tool.

We believe these results demonstrate that HADES is quite 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, the depth of the trees provided by the user are 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.

8. Related work

In recent years, program synthesis has received much attention from the programming languages community. Many of these approaches require either a programmer-provided “template” [7, 33, 34] or a complete logical specification [21] of the target program. In contrast, our method performs synthesis from examples.

There is a rapidly expanding literature on synthesis techniques that require only input-output examples. Many approaches in this area are focused on non-hierarchical data types like numbers [32], strings [15], tables [6, 16]. However, there is also an emerging body of work on the synthesis of programs over recursive data structures [5, 13, 23, 28]. Of these approaches, Perelman et al. [28] give a method based on optimized enumerative search for constructing program synthesizers for user-defined domain-specific languages (DSLs). Le and Gulwani [23] present a framework called FlashExtract where programs made from higher-order combinators are synthesized using a custom deductive procedure. Albarghouthi et al. [5] uses goal-directed enumerative search to synthesize first-order programs over recursive data structures. Feser et al. [13] offers a method to synthesize higher-order functional programs over recursive data structures using a combination of heuristics for generalization from examples, deduction and cost-directed enumeration. Osera and Zdancewic [27] study a similar problem and offer a solution based on type-directed deduction.

One difference between these approaches and ours is algorithmic. Unlike these approaches, our algorithm is based on a combination of SMT solving and decision tree learning. Also, these approaches aim to synthesize programs written in general programming languages or user-defined DSLs. In contrast, our approach is focused on synthesis of tree transformations and utilizes particular properties of such transformations. This tighter focus allows us to handle tree transformation benchmarks whose complexity exceeds benchmarks considered in prior work. For example, our method is able to synthesize transformations that alter the hierarchical structure of an input tree. So far as we know, such benchmarks fall outside the scope of prior approaches to example-driven synthesis.

Example-guided synthesis has a long history in the artificial intelligence community [19, 24, 25]. We build on the tradition of inductive programming [18, 20, 22], where the goal is to synthesize functional or logic programs from examples. Work here falls in two categories: those that inductively generalize examples into functions or relations [20], and those that search for implementations that fit the examples [17, 26]. These approaches are all based on various combinations of enumeration and deduction. In contrast, our approach is based on SMT solving and decision tree learning. Our subroutine for synthesizing path transformations has parallels with a strategy followed in Gulwani’s approach to synthesizing string transformations [15]. Like our algorithm, Gulwani’s technique also uses a combination of partitioning and unification. However, the algorithmic details are very different. For example, Gulwani’s approach 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.

Also related to this work are algorithms [9, 11, 12] for learning finite-state tree transducers [30] from examples. Applications of such algorithms include learning of XML queries and machine translation. 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 [11].

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 [31], identification of latent code errors [8], and repair of programs that manipulate relational databases [14]. So far as we know, we are the first to use decision tree learning in example-driven synthesis.

9. Conclusion

We have presented an algorithm for synthesizing hierarchical data transformations from examples. The central idea of our approach is to reduce the generation of tree transformations to the synthesis of transformations on paths of the tree. 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 also allowing us to handle a richer class of tree transformations, including those that change the data hierarchy.

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. In Automator, the visual notation is only an alternative syntax for deterministic, imperative programs, and this limits the complexity of programs that users can produce. HADES, which allows users to generate complex programs from simple examples, is a plausible way of broadening the scope of such tools.
References

Using the inductive hypothesis, we have that paths(T) = paths(T'). Since T ⊆ T', we have height(T) = height(T') = h. Let v = root(T) and v' = root(T'). Since T ⊆ T', we have L(v) = L(v') = ℓ 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' = ∅. Hence, paths(T) = paths(T') = {v, d}. For the inductive case, let h = k + 1 where k ≥ 1. Let C = children(v) and let C' = children(v'). Since paths(T) = paths(T') = {T} = C', and since T and T' are well-formed, there exists a one-to-one correspondence f : C → C' such that: f(vk) = v'k if f(vk) ∈ L(vk) = L(v'k). Using this fact and condition (3) of Definition 4, we know that, for every vk ∈ children(v) = C, subtree(T, vk) ≡ subtree(T', f(vk)). Now, observe that:

paths(T) = ∪

dwhicj

paths(T') is equal to:

paths(T') = ∪

Using the inductive hypothesis, we have paths(T) = paths(T').

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

paths(T) = ∪

Since paths(T) = paths(T'), this implies ℓ = ℓ' and d = d' and, since T, T' are well-formed, for each vk ∈ C, there must be a one-to-one correspondence f : C → C' such that v'k = f(vk) if f(vk) ∈ paths(T, vk) = paths(T', v'k). Now, for any pair (vk, f(vk)), we have L(vk) = L'(f(vk)) because paths(T, vk) = paths(T', v'k). Note that this implies condition (2) of Definition 4. Furthermore, since paths(T, vk) = paths(T', v'k), the inductive hypothesis implies subtree(T, vk) ≡ subtree(T', v'k). Hence, condition (3) of Definition 4 is also satisfied.

Proof of Theorem 2. Suppose TRANSFORM(T) = T'. We will show paths(T') = paths(T'), which implies T' ⊆ T' by Theorem 1. First, we show that, if p' ∈ paths(T'), then p' ∈ paths(T'). By the unambiguity criterion, there exists a unique p ∈ paths(T) such that p ∼ p'. By the correctness requirement for path transformer f, we know p' ∈ f(p) and p' ≳ p since p' ∈ paths(T'). Hence, p' ∈ S' in Figure 7. Since T' = SPLICE(S') and SPLICE guarantees that paths(T') = S', we also have p' ∈ paths(T'). Now, we show that, if p' ∈ paths(T'), then p' ∈ paths(T'). Since p' ∈ paths(T'), there must exist a p ∈ paths(T) such that p' ∈ f(p). Since p ∈ inputs(E), correctness of f implies there exists some (p, p') ∈ E. Now, suppose p' ∈ paths(T'). Since (p, p') ∈ E but p' ∉ paths(T'), there are two possibilities: (i) Either p' ∼ ⊥, or (ii) there is some other (T1, T2) ∈ E that results in p* getting added to E'. Now, (i) is not possible because paths(T*) cannot contain ⊥, and (ii) is not possible due to the unambiguity requirement.