Bachelorarbeit

Synthesizing List Transformations from Examples and its Usage for Hierarchical Data

Christian Klinger

March 22, 2016

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum
xx.xx - yy.yy

Gutachter
Prof. X

Betreuer
Dr. No

DECLARATION

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Abstract

This thesis presents a new algorithm to synthesize list transformations from input/output examples. This is an interesting problem because many transformations on tree structured data can be simulated by transformations on the path of the tree leaves.

During a research internship I worked on the HADES project with Navid Yaghmazadeh under the supervision of Prof. Isil Dillig at the University of Texas. The implementation of the presented algorithm is one of the key parts of the project.

I first give a short summary of the whole idea behind HADES, then I will present a basic version of the algorithm in full detail. Later I will discuss a few extensions of the algorithm to handle additional types of transformations. The last chapter contains an evaluation of HADES on real-world testcases.

Contents

Introduction

1.1 Program Synthesis

Program Synthesis describes the concept of *synthesizing* a program from a specification of the user. To some extent a *synthesizer* is similar to a compiler: Both take some specification from the user and both output some program. The main difference is that the specification for a compiler is usually some kind of higher language and translating high language into the target language is a rather mechanical task that follows the syntax of the input language.

Synthesizers, on the other hand, allow a huge variety of forms of specifications and the translation process is very non-trivial due to new arising problems such as under-specification or ambiguity or the problems that generally come with exponential search spaces.

Program Synthesis can serve different user groups. It allow users without programming knowledge to create programs. But also for programmers, synthesis can be a valuable tool within their workflow.

We can differentiate synthesizers by three properties a) the specification of the user's intent b) the space of programs which is searched and c) the search technique[4].

The oldest discipline is the synthesis of programs from formal specification. By giving a specification in first-order logic, it is theoretically possible to synthesize a program from a constructive proof. Early synthesizers were therefore based on theorem proving [6].

Another approach is *Programming by Example*. A user provides one or more input-output examples and the synthesizer is asked to synthesize a transformation which transforms each input into its corresponding output. The challenge with this is to find the most general transformation as it is easy to over-fit by synthesizing a transformation containing a big switch-statement encoding the mapping between input and out examples.

Close to *Programming by Example* is the idea of *Programming by Demonstration* where the user provides the synthesizer with a step-by-step instruction on how to transform the input to the output. Such an instruction contains more information than an input-output example but may require additional efforts from the user.

The perhaps most intuitive medium for a user to convey his intention might be natural language, making the problem even harder as understanding natural language is a hard problem in itself: Problems like semantic parsing have to be solved additionally to the synthesis.

As Gulwani noted[4], synthesizers can also be categorized by the search space over which programs are searched. Ideally, a synthesizer would search through a Turing-complete language. An example for this kind of synthesis is the "Superoptimizer" [7]. But "Superoptimizer" is also a good illustration why searching through a complete language is often unfeasible: "Superoptimizer" was unable to find any program consisting of more than 5 machine instructions. Even modern computing power wouldn't help much due to the exponential growth of the search space. This is why most recent attempts use a limited language. The challenge here, is to find the sweet spot between being synthesizable in practical time and a large functionality.

Depending on the provided specification, different search methods are available. Having a first-order logical specification a theorem prover can be used to derive a constructive proof which can be turned into a program. For other kinds of specifications SMT solving is sometimes applicable. The user's specification is translated into a formula (usually in first-order logic) and the assignment that satisfies the formula can be translated into a program.

Another interesting approach are version space algebras [8]. A version space is a compact representation of a set of functions that suffice some input/output examples. By using operators on these version spaces like union and intersection it is possible to build up the version space of the final program from smaller elements[5].

The synthesizer I will present in this paper – HADES with its underlaying list transformation synthesizer, uses *Programming by Examples*, searches a limited search space and uses a combination of backtracking and SMT solving.

1.2 Hades

Much of the data that users deal with today is inherently hierarchical or tree-based. Files, for example, are structured by file systems into trees, where directories are internal nodes and files are leaves. Same goes for XML documents where subtrees are identified by start and end tags. Another not too unpopular example for tree-shaped data are HDF files where groups correspond to internal nodes and datasets represent leaves.

Users of the data are often confronted with tasks that resemble *tree transformations*. A user might want to reorganize their file hierarchy or change the structure of an XML document. Usually users only have two options: Either to perform the transformation by hand or to write a program for this.

For most users the latter option is often not feasible, either because they are not programmers or because the task asks for a domain-specific language in which the programmer is not fluent. For example to re-organize files, bash-script would be a natural choice, whereas for transforming XML files a specific transformation language like XSL[2] or XQuery[1], would be better.

Motivated by these examples we developed a novel algorithm and implemented it in HADES, a tool that synthesizes transformations from input-output examples. HADES operates on an abstraction called *Hierarchical Data Trees*(HDTs). This allows for easy adaption to different domains (file systems, XML, ...). The only functionality that is domain-specific is a parser that parses the input-output example into an HDT and a code generator that generates code in the target language. So far we have implemented a file system plug-in that generates bash scripts and an XML plug-in that generates XSLT files.

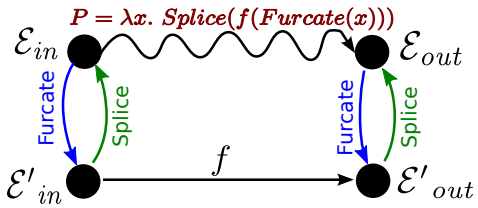


Figure 1.1: Synthesized programs use *Splice* and *Furcate* to simulate a tree transformation with a path transformation

Our algorithm places no restriction on tree depth or fanout and is able to synthesize a rich class of transformations that appear in real world tasks. We tested HADES on tasks collected from online forums with success.

1.2.1 Idea

Many tree transformations that appear in real world tasks can be expressed as a composition of path transformations, that is why all programs generated by HADES have a similar structure: 1. Take a tree as input 2. Generate a list of all paths 3. Apply the path transformation on each path 4. Combine the resulting paths into an output tree. The process in which a tree is decomposed into a set of paths will be called *furcation* and the inverse process of composing paths into a tree will be called *slicing*. Figure 1.1 illustrates the connection between a tree- and a path transformation.

A competing approach, which is arguably more straight-forward, would be to synthesize tree transformations directly [9, 3]. Programs generated by this approach are in general more compact, but their synthesis itself is very complex. Our approach, on the other hand, makes the synthesis easier. In other words: We trade off the complexity in the synthesis for more complexity in the generated program.

1.2.2 Overview

I will illustrate our approach by following an example from [11]. During the process we will see how Hades works. Before inferring path transformations, HADES first has to process the given input and output tree into pairs of paths so that these pairs can the be used as input-output examples to our path transformation synthesizer.

We say that a path transformation π suffices an example (p, p') if p' is a result of π applied on p.

To synthesize the path transformation we use a hybrid approach of decision tree learning and SMT solving. The SMT solver is used to find out whether a subset of the examples is "unifiable". Unifiable means in this context that all examples (p_i, p'_i) in the subset can be sufficed by one list transformation. Hades finds a minimal partitioning into unifiable subsets. After that, the decision tree algorithm is used to find classifiers for each of the partitions. The final path transformation is then basically a big "switch-statement" that uses the classifiers from the decision tree as predicates to execute the corresponding list transformations.

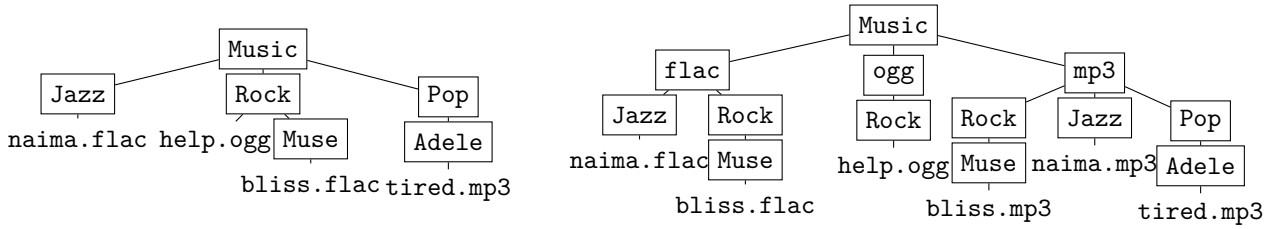


Figure 1.2: Input-output directories for motivating example

$\mathcal{E}_1 : p_1 = [(\text{Music}, d_m), (\text{Jazz}, d_j), (\text{naima}, d_s)] \mapsto p_1' = [(\text{Music}, d_m), (\text{flac}, d_f), (\text{Jazz}, d_j), (\text{naima}, d_s)]$

$\mathcal{E}_2 : p_1 = [(\text{Music}, d_m), (\text{Jazz}, d_j), (\text{naima}, d_s)] \mapsto p_1'' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Jazz}, d_j), (\text{naima}, d_s')]$

$\mathcal{E}_3 : p_2 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{help}, d_h)] \mapsto p_2' = [(\text{Music}, d_m), (\text{ogg}, d_o), (\text{Rock}, d_r), (\text{help}, d_h)]$

$\mathcal{E}_4 : p_3 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_u)] \mapsto p_3' = [(\text{Music}, d_m), (\text{flac}, d_f), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_u)]$

$\mathcal{E}_5 : p_3 = [(\text{Music}, d_m), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_u)] \mapsto p_3'' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Rock}, d_r), (\text{Muse}, d_{ms}), (\text{bliss}, d_u')]$

$\mathcal{E}_6 : p_4 = [(\text{Music}, d_m), (\text{Pop}, d_p), (\text{Adele}, d_a), (\text{tired}, d_{sf})] \mapsto p_4' = [(\text{Music}, d_m), (\text{mp3}, d_{mp}), (\text{Pop}, d_p), (\text{Adele}, d_a), (\text{tired}, d_{sf})]$

Figure 1.3: path transformation examples for Bob's files constructed by HADES

Example 1. Suppose there is a user Bob who owns a large collection of audio files. Bob wants to re-arrange the files into a different folder structure without moving every single file by hand.

Currently, his music folder is sorted by genre and artist regardless of the format of the file. For the new structure he wants to have three main folders `flac`, `ogg` and `mp3`. Under each of these folders, files should be placed under Genre and Artist. Files from the original library should be placed under one of the three main folders according to their extension. Additionally, Bob wants the flac-files to be converted to mp3 while still keeping the originals because his portable music player has no flac-support.

Since Bob's music collection is huge, it is not feasible for him to do the restructuring manually. Bob does not know how to program, so he cannot write a script either. Luckily for him, we developed HADES which he can use to synthesize a script in a matter of seconds.

To use HADES, Bob first creates a small example consisting just of a few files (see Figure 1.2). Bob calls HADES with these two folders as parameters. The file-system plug-in of HADES first converts these directories into *hierarchical data trees*(HDTs). A node in an HDT consists of a label l and some data d. A path is therefore a list of pairs (l_i, d_i) and begins always with the root node and ends in a leaf.

The trees are then furcated into two sets of paths. and corresponding paths are matched, such that we get a set of examples (p_i, p_i') such that p_i is a path of the input tree and p_i' is a path of the output tree and the leaves 'correspond' to each other. HADES itself is parametric to this notion of 'correspondence'. In our current implementation a user has to annotate corresponding leaves, but it is easy to imagine to use a fingerprinting technique.

Note that we allow paths to appear in more than one example in order to support duplication. For path p in the input tree with no corresponding path in the output tree we add the example $(p, \perp)$ to E to represent deletion.

In Figure 1.3 you can see the examples for the path transformation synthesis in Bob's case.

Having these examples, it is now time to find a partitioning such that for each partition we can find a unifying list transformation. We assume for now the existence of a procedure UNIFY (to which the whole next chapter is dedicated) which uses SMT solving to check whether such a list transformation exists.

HADES enumerates through the set of all partitionings in increasing order of size until a partitioning is found such that every partition in it is unifiable.

In Bob's case, we find two partitions $\mathcal{P}_1 = \{\mathcal{E}_1, \mathcal{E}_3, \mathcal{E}_4, \mathcal{E}_6\}$ and $\mathcal{P}_2 = \{\mathcal{E}_2, \mathcal{E}_5\}$. The unifying list transformations are

$$\begin{aligned}\chi_1 = \text{ListConcat } (\quad & \text{map Id subpath}(x, 0, 1), \\ & \text{map ExtOf subpath}(x, size(x) - 1, size(x)), \\ & \text{map Id subpath}(x, 1, size(x)))\end{aligned}$$

for $\mathcal{P}_1$ and

$$\begin{aligned}\chi_2 = \text{ListConcat } (\quad & \text{map Id subpath}(x, 0, 1) \\ & \text{"mp3"} \\ & \text{map FlacToMp3 subpath}(x, size(x) - 1, size(x)))\end{aligned}$$

for $\mathcal{P}_2$.

HADES then uses the ID3 algorithm[10] to find classifiers to separate the partitions from each other.

The classifier for $\mathcal{P}_1$ is just $\phi_1 : true$, because every input path in $\mathcal{E}$ is in $\mathcal{P}_1$. For $\mathcal{P}_2$ HADES infers the classifier $\phi_2 : \text{ext} = \text{"flac"}$. Finally, the complete path transformer is then

$$\pi : \lambda x.(\chi_1;\ \text{if(ext} = \text{"flac" then } \chi_2))$$

.

With this final path transformer π at hand, it is now the task of the file system plug-in to generate a program in the target language. In Bob's case, the plug-in produces a shell script (see Figure 1.4) that, when applied on his music library, structures it in the desired way.

```
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 1.4: structure of a synthesized bash script

In this thesis I am going to focus on the UNIFY procedure that synthesizes a list transformation for a set of examples and is one of the main algorithms in HADES. More information about HADES can be found in [11].

List Transformations

2.1 Overview

Synthesizing unifying list transformations is an integral part of HADES. In this chapter I will introduce a basic version of Unify. The current implementation in HADES contains a few extensions that either improve performance or extend the class of synthesizable transformations. These extensions are described in chapter 3.

The synthesis algorithm for list transformations was designed with the use case of tree transformations in mind. For that, we focus mainly on transformations that concatenate sublists of the input list. Other operations such as sorting or folding (as they are rarely needed for tree transformations) are not supported. We do allow the application of functions on list elements and the concatenation of labels to create new path labels.

The key idea is to represent examples and transformations in a numeric way. This allows efficient processing with an SMT solver.

To illustrate how Unify works, I will follow an example from the file system domain. As already mentioned, a path element consists of a label l and the data d. To build some intuition about what kind of transformations we can handle, I would like to neglect the data part for now and just focus on the labels.

Let's assume we want to unify the following two examples: A file `a.jpg` which is stored under `/photos/football/2015/03/good/a.jpg` was moved to `/photos/201503/good/a.jpg` and a second file `b.jpg` was moved from `/photos/baseball/2014/01/b.jpg` to `/photos/201401/b.jpg`.

We would get the following examples (pairs of lists).

$$E = \{([\texttt{photos}, \texttt{football}, 2015, 03, \texttt{good}, \texttt{a.jpg}], [\texttt{photos}, 201503, \texttt{good}, \texttt{a.jpg}]),\\ ([\texttt{photos}, \texttt{baseball}, 2014, 01, \texttt{b.jpg}], [\texttt{photos}, 201401, \texttt{b.jpg}])\}$$

Unify begins with *summarizing* the examples, that means that for each example we find a trivial transformation that suffices it. Through later steps we will find ways to gradually make them more similar until we can decide whether these transformations are unifiable and if unifiable can easily deduce a unifying transformation.

When we look at the output of the first pair: [`photos`, `2015-03`, `good`, `a.jpg`] we can build a 'trivial' transformation by concatenating sublists of size 1.

- `photos` is nothing else then the sublist from 0 to 1,
- `201503` is a string concatenation of two labels, and so forth.
- `good` is the sublist from 4 to 5
- ...

So the trivial transformation of the first example is

$$\begin{aligned} T_1 = \text{ListConcat} (\quad & \text{map Id } (0,1), \\ & \text{map stringConcat}(i, i+1)(2,3), \\ & \text{map Id } (4,5), \\ & \text{map Id } (5,6)) \end{aligned}$$

and the trivial transformation of the second example is

$$\begin{aligned} T_2 = \text{ListConcat} (\quad & \text{map Id } (0,1), \\ & \text{map stringConcat}(i, i+1)(2,3), \\ & \text{map Id } (4,5)) \end{aligned}$$

Through later steps we will also need to know the size of input lists. We will refer with the term "summarized example" to pairs of input length and trivial transformation.

$$E' = \{(6, T_1), (5, T_2)\}$$

To decide whether T_1 and T_2 are unifiable we need them to have the same structure, meaning that they have the same number of elements in the ListConcat with the same functions (Id, stringConcat).

For that we can *coalesce* the transformation T_2, joining the map Id$(4,5)$ with map Id$(5,6)$. Our two *summarized and coalesced examples* transformations are then

$$E^* = \quad \{ \quad \begin{array}{l} (6, \text{ListConcat} (\quad \text{map Id } (0,1), \quad \text{map stringConcat } (i, i+1)(2,3), \quad \text{map Id } (4,6))), \\ (5, \text{ListConcat} (\quad \text{map Id } (0,1), \quad \text{map stringConcat } (i, i+1)(2,3), \quad \text{map Id } (4,5))) \quad \} \end{array}$$

Unifying these two examples is now child's play. We can easily deduce that a unifying list transformation also has 3 elements in the listConcat and that the first two elements are just the same as in our example. For the third segment we would have to find a unifying term that evaluates to 6 for the first example and to 5 for the second example. In our example the term would just be $1 \cdot \text{size}(x)$. (This is where the SMT solver gets involved)

This example illustrated how each example got summarized and coalesced and how the examples together were unified. This example is designed in a way that each operation is unambiguous. In reality there are often multiple ways to summarize an example, some examples can also be coalesced but shouldn't in order to be unifiable. To deal with these sources of ambiguity it makes sense to approach the problem a little more formal.

List transformation χ	:=	ListConcat$(\tau_1(x), \ldots, \tau_n(x))$
Segment trans. τ	:=	$\lambda x.$ map $F(x)$ (t_1, t_2)
Index term t	:=	$b \cdot \text{size}(x) + c$
Mapper function F	:=	$\lambda x.\lambda i. \ldots$

Figure 2.1: Language for describing list transformations

$$\chi_1 = [\langle 0, 1,\ Id\rangle,\ \ \langle (size(x) - 1, size(x),\ ExtOf\rangle,\ \ \langle 1, size(x),\ Id\rangle]$$
$$\chi_2 = [\langle 0, 1,\ Id\rangle,\ \ "mp3",\ \ \langle size(x) - 1, size(x),\ FlacToMp3\rangle$$

Figure 2.2: introductory example in short-hand notation

2.2 Unify Algorithm

Definition 1 (List Transformation)**.** The language of list transformations can be seen in Figure 2.1. Each list transformation consists of a set of segment transformers. Segment transformers apply a mapper function over a range of the input list. The range (t_1, t_2) includes the first element but excludes the second. The range has to be specified with index terms of the form $b \cdot size(x) + c$ where b is a binary coefficient and c is an integer.

For brevity I want to introduce a short-hand notation which will be used thorough this document. Instead of map F (t_1, t_2) I will write $\langle t_1, t_2,\ F\rangle$. In a similar fashion ListConcat $(\tau_1, \tau_2, \ldots, \tau_n)$ will be shortened to $[\tau_1, \tau_2, \ldots, \tau_n]$. The transformation of the Introduction (Bob's files) in short-hand notation is illustrated in Figure 2.2.

Mapper functions are functions that transform a node into its corresponding node. The mapper functions change label and data of the node. Instead of passing the node x_i itself, we pass the input path x and the position i to a mapper function to allow for transformations based on path elements other than x_i. An example for a mapper function is "append the name of the parent node and set the owner to 'max'". Mapper functions might also ignore their input and just 'create' a new node, for example "node 'pictures' with owner 'max' and permissions 777".

The language of mapper functions can be seen in Figure 2.3. Each mapper function consists of a stringConcat element that describes the change in the label of the node as a concatenation of strings and a dataTransformer which maps changes in the data of the node.

The dataTransformers are only inferred after the UNIFY algorithm has found a unification for the indexTerms and the stringConcat. For HADES the stringConcat are also the more interesting part, because the labeling determines the tree structure.

A StringConcat concatenates strings from different origins, whereas the strings are denoted by a numerical term s. These terms can refer to strings of three different origins:

Case 1: Positional When we want to refer to the label of node x_i, we just use a value of $s = i$ as an element in the stringConcat.

$$\begin{array}{lcl} \text{Mapper function } F & := & (\text{StringConcat}, \text{dataTransformer}) \\ \text{StringConcat} & := & \lambda x \lambda i \ . \ concat(s_1, s_2, \ldots s_n) \\ & & \quad \text{where } s = x[b \cdot i + c].l \end{array}$$

Figure 2.3: language for mapper functions

Case 2: Dictionary When want to refer to a constant string, we use indices in a high range to identify entries in a global dictionary. For brevity, we will just assume that 1000 is high enough. For a reason that will become clear later, we assign only non-consecutive indices. The first dictionary word is then $s = 1001$, the second $s = 1003$ et cetera.

Case 3: Function Application The functions which can be applied, $\mathcal{F}$, extract data from the nodes. For example in the file-system domain, we have functions that return the owner, the group, permissions, modification dates etc. For XML we have one function for each occurring attribute in the XML file. Each of these functions also gets a unique range of numbers that are neither used by Case 1 nor Case 2. For brevity we will just assign 11000 to the first function, 12000 to the second, et cetera.

We will store the assignments in a mapping $\mathcal{D}$.

$$\mathcal{D} = \{\text{extension} \to 11000, \text{owner} \to 12000, \text{group} \to 13000, \text{modification\_year} \to 14000, \ldots\}$$

A function f applied to node x_i is now referred to as $s = \mathcal{D}[f] + i$, which is also just a number.

In the following section I want to define the UNFIY procedure in terms of three sub-algorithms SUMMARIZE, COALESCE and SIMPLEUNIFY.

2.2.1 Summarization

Given an example $\mathcal{E} = (x, x')$, the summarization of $\mathcal{E}$ is a set $\mathcal{E}' = \{(size(x), T) \mid T \in T^*\}$ where $size(x)$ is the number of elements in x and T is a set of trivial transformations that transform x into x'.

To create the transformations we rely on a procedure MAKESEGMENTTRANSFORMERS. MAKESEGMENTTRANSFORMERS takes a single list element x'_i and creates segment transformers that, when applied on x return the element x'_i. We apply this procedure on every element in the example output. A trivial transformation is then just a concatenation of all those segment transformers.

Since MAKESEGMENTTRANSFORMERS may return more than one transformer, we calculate a Cartesian product to capture all possible list transformations which transform x into x'.

To illustrate the work of Summarize, let us make an example. Suppose we get the example $\mathcal{E}_1 = ([a, b, a], [a\_b, b])$. In this case there are three different segment transformers that create a_b.

Algorithm 2 Summarize

procedure SUMMARIZE(example $\mathcal{E} = (x, x')$)
  for all $i \in [0 \dots \text{size}(x')]$ **do**
    $\mathcal{T}_i :=$ MAKESEGMENTTRANSFORMERS$(x, x'[i])$
  end for
  $T^* := \{\text{ListConcat}(\tau_1, \tau_2, \dots \tau_{\text{size}(x')} \mid \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2, \dots \tau_{\text{size}(x')} \in \mathcal{T}_{\text{size}(x')}\}$
  return $\mathcal{E}' = \{(\text{size}(x), T) \mid T \in T^*\}$
end procedure

MAKESEGMENTTRANSFORMERS just adds the string 'a_b' to the dictionary. Let's assume the string gets the unique dictionary index 1001. and returns the segment transformer

$$\langle 0, 1, \text{ stringConcat } (1001)\rangle$$

We can get another segment transformer if we add '_' to the dictionary (assuming index 1003) and describe the string 'a_b' as a concatenating of label 0 and '_' and label 2.

$$\langle 0, 1, \text{ stringConcat } (i, 1003, i+1)\rangle$$

The third segment transformer is similar to the second one but draws the substring 'a' from the third label.

$$\langle 2, 3, \text{ stringConcat } (i, 1003, i-1)\rangle$$

For the second label in the output of the example ('b'), MAKESEGMENTTRANSFORMERS can only produce one segment transformer:

$$\langle 1, 2, \text{ stringConcat } (i)\rangle$$

The complete summarization is then a set of all combinations of a segment transformer for the first output element and of a segment transformer for the second output element (for which there is only one).

$$\begin{aligned}\text{SUMMARIZE}([a, b, a], [a\_b, b]) = \\ &(3, [\langle 0, 1, \text{ stringConcat } (1001)\rangle, \langle 1, 2, \text{ stringConcat } (i)\rangle]),\\ &(3, [\langle 0, 1, \text{ stringConcat } (i, 1003, i+1)\rangle, \langle 1, 2, \text{ stringConcat } (i)\rangle]),\\ &(3, [\langle 2, 3, \text{ stringConcat } (i, 1003, i-1)\rangle, \langle 1, 2, \text{ stringConcat } (i)\rangle])\end{aligned}$$

Normalization A reader might ask why I wrote one segment transformer as $\langle 2, 3, \text{ stringConcat } (i, 1003, i-1)\rangle$" as $2, 3$ and not as $\langle 0, 1, \text{ stringConcat } (i+2, 1003, i+1)\rangle$. The reason is that we can reduce the number of possible mapper functions and therefore increase the performance of the overall algorithm by normalization. Consider the two segment transformers:

$$\tau_1 = \langle 1, 2, \text{ stringConcat } (i)\rangle \qquad \tau_2 = \langle 2, 3, \text{ stringConcat (i-1)}\rangle$$

It's easy to see that τ_1 and τ_2 always produce the same output. To avoid the necessity to consider all of these candidate segment transformers we will stipulate that in the first concatElement that does not refer to a dictionary word (remember index terms are of the form $b \cdot i + c$) will always

have a $c = 0$, hence τ_1 is in normal form, but τ_2 is not.

2.2.2 Coalescence

Through summarization we converted the examples into list transformations. Our main goal is to rewrite these transformations until we find a unifying list transformation. For now, the main difference in the structure of these list transformations are their different lengths. Therefore we introduce a function COALESCE (Algorithm 3) which can change the length of the list transformation of the summarized examples transformation without changing its semantics.

COALESCE returns a set that additionally to the original summarized examples also contains summarized examples where the transformations are shortened. The shortening works by merging adjacent segments transformers which have the same mapper function.

For example: if we had a summarized example $\mathcal{E}' = \{(2, [\langle 1, 2, \text{ stringConcat } i\rangle, \langle 2, 3, \text{ stringConcat } i\rangle])\}$ then the coalesced example $\mathcal{E}^*$ would additionally contain $(2, \langle 1, 3, \text{ stringConcat } i\rangle)$.

Algorithm 3 COALESCE

Inference rules to coalesce a summarized example $\mathcal{E}'$ into $\mathcal{E}^*$.

$$\frac{(n, T) \in \mathcal{E}'}{(n, T) \in \mathcal{E}^*} \tag{2.1}$$

$$\frac{(n, [\ldots, \langle t_1, t_2, F\rangle, \langle t_2, t_3, F\rangle, \ldots]) \in \mathcal{E}^*}{(n, [\ldots, \langle t_1, t_3, F\rangle, \ldots]) \in \mathcal{E}^*} \tag{2.2}$$

2.2.3 Simple Unification

Definition 2 (Simple Unifiability)**.** A set of summarized (and coalesced) examples

$$\bar{E} = \{(n_1, T_1), (n_2, T_2), \ldots, (n_N, T_N)\}$$

whose transformations are of length l is *simply unifiable* if there exists a transformation $\chi = \left[\tau_1, \tau_2, \ldots \tau_l\right]$ such that for each example (n_i, T_i) the substitution of size(x) with n_i results in the transformation T_i.

$$\forall_{i \in [1 \ldots N]} \quad \chi[n_i/\text{size}(x)] = T_i \tag{2.3}$$

We call such a χ unifier of $\bar{E}$.

Finding a unifier is easier than the reader might think; most of the work has already been done by summarizing and coalescending the examples.

Observe that the transformations in $\bar{E}$ do not use the full domain of transformations: Due to their construction all terms in transformations of $\bar{E}$ denoting begin and end of the sublists are numeric constants. Our language permits terms of the form $b \cdot size(x) + c$, this gives us space to find unifying terms.

Consider the following two summarized and coalesced examples:

$$T_1 = (4, [\langle 1, 3,\ F_1\rangle, \langle 3, 4,\ F_2\rangle]),$$
$$T_2 = (5, [\langle 1, 3,\ F_1\rangle, \langle 4, 5,\ F_2\rangle])$$

We can straight-away reason that our unifier also has to have two segments, with F_1 and F_2 as their mapper functions.

$$\chi_h = [\langle t_1, t_1',\ F_1\rangle, \langle t_2, t_2',\ F_2\rangle]$$

Sometimes we call this formula a *hypothesis* with *holes* $t_1, t_1', t_2, t_2', \ldots$. Simple unification is then done by finding terms of the form $b \cdot size(x) + c$ such that (2.3) is satisfied. For this we would use the SMT solver. In our example, values for t_1 and t_1' are trivially to find as they are already the same in the two transformations.

For the hole t_2 we have the following constraints:

$$t_2[4/size(x)] = 3 \qquad \wedge \qquad t_2[5/size(x)] = 4$$

Using the SMT solver we get the solution: $t_2 = 1 \cdot size(x) + (-1)$. Similarly for t_2' the solution is $1 \cdot size(x) + 0$.

The final unifier (hypothesis with resolved holes) is then

$$\chi = [\langle 1, 3,\ F_2\rangle, \langle n-1, n,\ F_2\rangle]$$

As mentioned before, the task of finding terms to fill the holes is solved with an SMT solver. In the following paragraph I will describe an encoding of the 'simple unification' problem as an SMT formula.

From our first definition of simple unifiability and the fact that two segment transformers are equal when they have the same index terms (

SMT formula specifying χ

Through the coalescence step we made sure that all sets of summarized examples for which we call SimpleUnify have the same number of segment transformers l. Because of that we can also create our hypothesis χ_h with size l.

$$\chi = \left[\langle t_1, t_1', F_1\rangle, \langle t_2, t_2', F_2\rangle, \ldots, \langle t_l, t_l', F_l\rangle\right]$$

Each hole t_i and t_i' must be filled with a term of the form

$$t_i = b_i \cdot n + c_i \qquad\qquad t_i' = b_i' \cdot n + c_i'$$

Finding a unifier is now precisely reduced to finding these b_i and c_i. Let $\text{begin}(e, i)$ be the first argument of the i-th segment transformer in example e and analogous $\text{end}(e, i)$ the second argument and $\text{transformation}(e, i)$ the third argument. Then ϕ is the corresponding SMT formula to the goal defined in equation 2.3.

If ϕ is satisfiable, the list transformations are unifiable and the model of ϕ gives us an assignment to the holes of our hypothesis χ_h that turns it into a unifying list transformation.

For any set $\bar{E}$ of N summarized and coalesced examples whose transformations are of length l, the corresponding SMT formula is ϕ.

$$\phi = \bigwedge_{i=1}^{N} \bigwedge_{j=1}^{l} (t_{i,j} \wedge t'_{i,j})$$
$$t_{i,j} = b_i \cdot n_i + c_i = \text{begin}(e_i, j)$$
$$t'_{i,j} = b'_i \cdot n_i + c'_i = \text{end}(e_i, j)$$

Algorithm 4 shows the SIMPLEUNIFY procedure.

Algorithm 4 SIMPLEUNIFY

procedure SIMPLEUNIFY(set of examples $\bar{E}$)
  if $\bar{E}$ contains two examples e, e' that differ in the length or mapper functions **then**
    return failure
  end if
  let $\phi = \bigwedge_{i=1}^{N} \bigwedge_{j=1}^{l} (t_{i,j} \wedge t'_{i,j})$
  if SAT(ϕ) **then**
    get assignment α satisfying ϕ
    let $T = [\langle t_1^\alpha, t_1'^\alpha,\ F_1\rangle, \langle t_2^\alpha, t_2'^\alpha,\ F_2\rangle], \ldots \langle t_l^\alpha, t_l'^\alpha,\ F_l\rangle]$
      where $F_i = \text{transformation}(e_1, i)$
  else
    return failure
  end if
end procedure

2.2.4 UNIFY in terms of SUMMARIZE, COALESCE and SIMPLEUNIFY

After explaining every component in detail, it is time to have a look at the bigger picture again. To discuss how the three sub-algorithms SUMMARIZE, COALESCE and SIMPLEUNIFY can be combined into our desired UNIFY algorithm it makes sense to recapitulate the procedures in terms of their types.

- SUMMARIZE takes one example (which is a pair of paths) and returns a set of *summarized examples* (which itself are pairs of a number and a list transformation).
- COALESCE takes a set of summarized examples and returns a set of summarized examples.
- SIMPLEUNIFY takes a set of summarized examples and either returns a list transformation or "failure".

UNIFY (Algorithm 5) is supposed to take a set of examples $\mathcal{E}$ and return one list transformation that suffices all of the examples in the set. An example is a pair of paths, so we apply SUMMARIZE to each of the examples to get a set of sets of summarized examples.

Each of the sets contains different representations of the same example, but because we also want each set to contain equal representations of different lengths we also apply COALESCE to each of the sets. After that, $\mathcal{E}^*$ is still a set of sets of summarized examples.

Then we want to call SIMPLEUNIFY on sets that contain one summarized example per original example. Unfortunately (performance-wise) we have to consider all combinations, that's why we take the Cartesian product of $\mathcal{E}^*$.

When we find one unifiable set in the Cartesian product $\bar{\mathcal{E}}$ we can end our search and return the unifying list transformation χ. When, after considering all sets in $\bar{\mathcal{E}}$, we still have not found a unifier, we can signal that there is no unifier for the given examples $\mathcal{E}$.

Algorithm 5 UNIFY

```
procedure UNIFY(examples E)
    let  E'  = map SUMMARIZE E
         E*  = map COALESCE E'
         Ē   = Cartesian(E*)
    for all e in Ē do
        listTransformation χ = SIMPLEUNIFY(e)
        if χ ≠ FAILURE then
            return χ
        end if
    end for
    return "Not unifiable"
end procedure
```

Implementation

The presented list synthesis algorithm is (in an extended version) implemented as part of the HADES tool. HADES is implemented in C++ with approximately 9000 lines of code. In this section I want to showcase a few implementation details, either because they are important for the performance of the algorithm or because they increase functionality over the presented algorithm.

3.1 Lower and Upper Limits for the size of χ

We call UNIFY for each set in the Cartesian product ($\prod \mathcal{E}^*$) but we also know that UNIFY will fail for any set of in which examples have different lengths. This means we effectively prune this search space. Consider R as the set description of the pruned search space and let $L(\{e_1, e_2, \ldots, e_n\}) = \{\text{length}(\text{transformation}(\mathcal{E}_i)) \mid 1 \leq i \leq n\}$.

$$R = \{\{e_1, e_2, \ldots, e_N\} \mid e_1 \in \mathcal{E}_1^*, e_2 \in \mathcal{E}_2^*, \ldots, e_N \in \mathcal{E}_N^* \wedge \text{length}(e_1) = \text{length}(e_2) = \cdots = \text{length}(e_N)\}$$

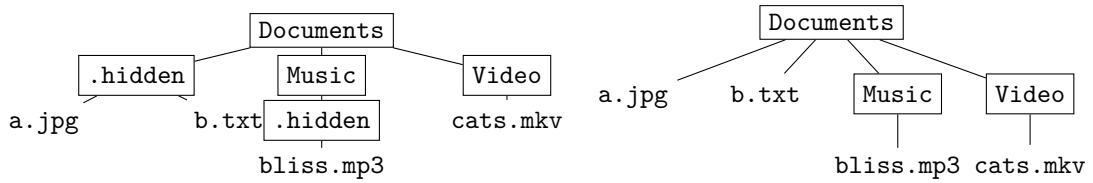
This implies that

$$\text{length}(\chi) \geq \max_{i \in 1..N} (\min L(\mathcal{E}_i^*))$$
$$\text{length}(\chi) \leq \min_{i \in 1..N} (\max L(\mathcal{E}_i^*))$$

Together with an efficient implementation of COALESCEN which calculates directly coalesced forms of length n without having to coalesced *all* forms we can use this to perform synthesize in a much more efficient way.

3.2 Lifting the example restriction

For the basic algorithm we had an implicit restriction. The algorithm required that every example contains at least one element that correspond to a segment transformer in the unifier. In other words, when the unifier is executed on the example inputs, none of the segment transformers is allowed to evaluate to an empty list.



(a) Original file structure

(b) Bob's desired file structure

Figure 3.1: Transformation: $[\langle 0, \text{indexOf}(.hidden),\ IdM\rangle, \langle \text{indexOf}(.hidden)+1, n,\ IdM\rangle]$ which moves all contents of a ".hidden" folder one level up.

We found out that we can lift this restriction at lower costs than expected. To relax the restriction we added a third rule to the coalesce algorithm (see Algorithm 3):

$$\frac{[\ldots,\ldots] \in \mathcal{E}^*}{[\ldots,\epsilon,\ldots] \in \mathcal{E}^*}$$

The direct consequence of this rule is that $\mathcal{E}^*$ grows to infinite size. This has direct implications on the lower bound on the length of χ. The lower bounds gets lowered to

$$length(\chi) = \min_{i \in 1..N} (\min \mathcal{E}_i^*)$$

Fortunately the upper bound is still valid.

Another consequence is that we have to change our SMT formula a little. If the i-th segment transformer of an example e is empty we replace the constraints $t_i = \text{begin}(e, i)$ and $t'_i = \text{end}(e, i)$ with more the more relaxed constraint $t_i = t'_i$.

3.3 Using indexOf variables

Another feature we added to HADES are what we call *indexOf*-variables. Until now, terms were of the form $b \cdot \text{size}(x) + c$. Sometimes this is not enough. There is a class of problems which are solvable if we had terms like $b_1 \cdot \text{size}(x) + b_2 \cdot \text{indexOf}(s_1) + b_3 \cdot \text{indexOf}(s_2) \cdots + c$.

Implementation of this was fairly simple. For a list x with elements x_i we define a function $pos(x, x_i) = i$. Remember that SUMMARIZE of an example (x, x') used to return the pair (n, T). Our new SUMMARIZE' returns now objects of the form (A, T), where A is a mapping from variable names to values.

$$A = \{n \to size(x), index_{x_1} \to pos(x, x_1), \ldots index_{s_n} \to pos(x, x_n)\}$$

Other functions like COALESCE are then just rewritten to pass $\text{Pos}(p)$ along n until both arrive at the place where the SMT formula is built.

An example what we can do it HADES with the indexOf variables is shown in Figure 3.1.

3.4 Minimality

An important goal in programs synthesis is to find the *most general* transformation, meaning that it is applicable to a wider range of inputs. HADES already ensures that the number of branches is minimal, but there are two more features that we want to minimize.

First of all, we want to have the minimal number of segment transformers per branch. For that, we (conceptually) sort the set $\bar{\mathcal{E}}$ in algorithm UNIFY by the number of segment transformers before we iterate over it.

In practice we don't actually need to sort $\bar{\mathcal{E}}$ because we use a modified lazy-evaluated implementation of COALESCE that calculates the result beginning from the transformation with the least number of segment transformers.

Secondly, we also want to have a minimum number of non-zero coefficients. Consider the index terms $1 \cdot \text{size}(x) + 1 \cdot \text{indexOf}('foo') - 3$ and $1 \cdot \text{sizeOf}(x)$. We can assume that for a given set of examples there were two unifying list transformations which differ in these index terms. (e.g. an example set where in all input paths 'foo' appears on the third index).

We could argue that the ladder term is more general, because it makes less assumptions about the shape of the input, for example it doesn't require that the label 'foo' appears at all. That's why we want to minimize the number of non-zero coefficients.

To enforce that condition we define a cost variable C as the sum over all coefficients within the SMT context. Then we perform a small (binary) search by adding and removing equations like $C < 5$, $C - 3$, ... and checking for satisfiability.

Evaluation

To evaluate HADES we collected 36 tasks in the file system and XML domain from online forums (stackoverflow.com and bashscript.org) and from teaching assistants at UT.

As with most synthesizers, the performance significantly depends on the quality of the user input. To make sure that we do not introduce some 'expert bias', we conducted a small user study with 6 students of which neither had any prior knowledge of the tool. After giving them an introduction about how to use HADES we asked them to solve the selected tasks. For detail on the user study see [11] for details. Figure 4.1 and Figure 4.2 contain descriptions of the tasks.

For the performance experiments we used examples from the user study. HADES was executed on a laptop with an i7-4800MQ and 8GB DDR3 memory running Debian Linux. Figure 4.3 shows the performance of HADES for the file system benchmarks. The tasks required programs with 1-4 branches and typically 2-5 segments. Over 90% of the benchmarks were synthesized in under a second.

The last two columns show the number of calls to the Unify and the number of queries to the SMT solver. Clearly, there is a correspondence between the number of branches and examples with the number of calls to Unify as the procedure is called many times when HADES tries to find a minimal partitioning to check whether a partition is unifiable.

In many cases, the number of SMT instances is lower or equal to the number of calls to Unify, this happens when the Summarize and Coalesce have an unambiguous result. In other cases, like F13, these methods return a multitude of summarized and coalesced examples of which all combinations have to be considered, leading to an increase in SMT formulas that have to be checked. This can usually be traced back to a less than optimal input from the user but was expected for a system targeted at non-experts.

F01 Given a folder with several .csv files, place each .csv file into a folder named after the group owner of the file.

F02 Given a folder with different files distributed over several folders, make all files with the extension ".sh" executable

F03 Find all text and bash files and opy them to a directory temp

F04 Given a folder structure where all files are placed at least three level deeps in subfolders, flatten the hierarchy by appending the last 3 directory names to the file name and delete directories.

F05 Put files in directories based on modification year/month/day

F06 Copy files without extension into the "NoExtension" directory

F07 Archive each directory to a tarball with modify month in its name

F08 Make files in "DoNotModify" directory read-only

F09 Convert .mp3, .wma, and .m4a files to .ogg but keep the directory structure intact

F10 Change group of text files in "Public" directory to "everyone"

F11 Change directory structure of ACM contest submissions in this way: `./contest-dir/submissions/{user-id}/{prob-id}/{subm-id}/CodeFile.cpp` to `./contest-dir/submissions/{prob-id}/{subm-id}-{user-id}/CodeFile.cpp`

F12 Convert all .zip archives in the directory to tarballs

F13 Organize all files based on their extensions

F14 Append modification date to the file name

F15 Convert pdf files to swf files

F16 Delete all files which were not modified last month

F17 Convert video files to audio files and put them in "Audio" directory

F18 Convert xml files that have a size greater than 1kB to text files

F19 Append "largest" to the name of the largest file and "big" to xml files which are bigger than 1 kB within every subdirectory

F20 Extract tarballs to a directory named using file and parent directory

F21 Append parent name to each .c file and duplicate it into a folder "MOSS"

F22 Copy a directory but only keep all files older than 5 days

F23 Copy each file to the directory created with its file name

F24 Create zip-files for directories that are not older than a week.

Figure 4.1: Description of all file system benchmarks

X01 Given an HTML document, add `style="bold"` to parent element of each text node.

X02 We have an XML file which consists of multiple `<group>` elements. Inside each of the groups, we have different `<person>` elements (which might be grouped into subgroups). Each of the `<person>` tags has an attribute `status`. The script should re-group the persons into groups with the same status where the `status` attribute gets moved from the person tag to the group tag.

X03 Remove all attributes of all elements but keep the document structure intact.

X04 Replace the document's root element from "xml" to "html".

X05 Remove third element and put all nested elements under parent

X06 For any XML document, create an HTML document containing a table in standard HTML (`<table>`, `<tr>`, `<td>`) that maps each text element to its parent element tag.

X07 Given an XML file where parts of the tree are contained in `<done>`, `<started>` or `<inProgress>`, generate a todo-list that contains the inner text of elements that were in either the `<started>` or `<inProgress>` tag.

X08 Generate an HTML drop down list with `<select>` and `<option>` from an XML document that contains the data in a more nested format.

X09 Rename a set of element tags to standard HTML tags.

X10 Suppose we have an XML file with `<item>` elements nested in `<elem>` elements. Each `<elem>` can have an attribute `"class"` with the values `missing` or `available`. We want to put all items in the same class under the same parent move the class attribute to the parent.

X11 Take an XML document that contains elements like `<item tag="p">foo</item>` and output an HTML document where these elements were converted into proper tags like `<p>foo</p>`.

X12 Delete all elements with tag `<p>` but keep the contents of it.

Figure 4.2: Description of XML benchmarks

Benchmark	Time(s)	Branches	Segments	Examples	Depth	# Unify	# SMT
F01	0.02	1	2	2	3	1	1
F02	0.03	2	2	2	2	5	5
F03	0.03	2	3	4	3	6	4
F04	0.01	1	2	2	6	1	1
F05	0.01	1	4	2	4	1	2
F06	0.03	2	3	7	4	9	4
F07	0.01	1	1	2	2	1	1
F08	0.09	2	2	4	3	14	14
F09	0.01	1	1	3	3	1	1
F10	0.04	2	2	3	2	7	7
F11	0.01	1	4	3	6	1	1
F12	0.06	2	2	3	3	6	6
F13	0.91	4	10	9	3	51	1567
F14	0.01	1	2	2	3	1	1
F15	0.01	2	2	2	2	4	2
F16	0.03	2	2	5	2	7	3
F17	0.01	1	2	3	5	1	1
F18	0.07	2	2	4	2	11	11
F19	11.33	3	5	6	3	19	27571
F20	0.33	2	3	3	3	5	171
F21	0.04	2	3	4	4	6	4
F22	0.06	2	2	7	2	9	6
F23	0.03	1	2	3	5	1	1
F24	0.11	2	2	5	2	12	12

Figure 4.3: Benchmark results for the file system domain

Benchmark	Time(s)	Branches	Segments	Examples	Depth	# Unify	# SMT
X01	0.01	1	2	3	5	1	1
X02	0.01	1	3	3	5	1	1
X03	0.01	1	1	3	3	1	1
X04	0.01	1	2	2	3	1	1
X05	0.01	1	2	3	4	1	1
X06	0.07	2	10	6	4	8	8
X07	0.04	2	3	8	3	10	4
X08	0.01	1	4	3	3	1	1
X09	0.01	1	4	2	3	1	1
X10	0.13	2	6	6	3	17	17
X11	3.83	3	9	6	3	19	16519
X12	0.02	2	1	5	4	7	3

Figure 4.4: Benchmark results for the XML domain

Bibliography

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An xml query language, 2002.

[2] J. Clark et al. Xsl transformations (xslt). *World Wide Web Consortium (W3C). URL http://www. w3. org/TR/xslt*, page 103, 1999.

[3] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–239. ACM, 2015.

[4] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

[5] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[6] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[7] H. Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGPLAN Notices*, volume 22, pages 122–126. IEEE Computer Society Press, 1987.

[8] T. M. Mitchell. An analysis of generalization as a search problem. In *Proceedings of the 6th international joint conference on Artificial intelligence-Volume 1*, pages 577–582. Morgan Kaufmann Publishers Inc., 1979.

[9] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 619–630. ACM, 2015.

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

[11] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. *PLDI*, 2016.