Test-Driven Synthesis

Daniel Perelman *
University of Washington
perelman@cs.washington.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Dan Grossman
University of Washington
djg@cs.washington.edu

Peter Provost
Microsoft Corporation
peterpr@microsoft.com

Abstract

Programming-by-example technologies empower end-users to create simple programs merely by providing input/output examples. Existing systems are designed around solvers specialized for a specific set of data types or domain-specific language (DSL). We present a program synthesizer which can be parameterized by an arbitrary DSL that may contain conditionals and loops and therefore is able to synthesize programs in any domain. In order to use our synthesizer, the user provides a sequence of increasingly sophisticated input/output examples along with an expert-written DSL definition. These two inputs correspond to the two key ideas that allow our synthesizer to work in arbitrary domains. First, we developed a novel iterative synthesis technique inspired by test-driven development—which also gives our technique the name of *test-driven synthesis*—where the input/output examples are consumed one at a time as the program is refined. Second, the DSL allows our system to take an efficient component-based approach to enumerating possible programs. We present applications of our synthesis methodology to end-user programming for transformations over strings, XML, and table layouts. We compare our synthesizer on these applications to state-of-the-art DSL-specific synthesizers as well to the general purpose synthesizer Sketch.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming—Program synthesis

General Terms Languages, Experimentation

Keywords program synthesis, end-user programming, test driven development

1. Introduction

Programming-by-example (PBE [4, 20]) empowers end-users without programming experience to automate tasks like normalizing table layouts or reformatting strings merely by providing input/output examples. Present PBE technologies are designed for a specific set of data types usually either by using an SMT solver or similar technology [30, 33] and being limited to efficiently handling only those data types whose operations map well to the SMT solver's theories or by creating a domain-specific language (DSL) for the task in concert withs a program synthesis algorithm capable of producing programs in that DSL [8, 11, 18]. LaSy[1] instead is able to synthesize programs in arbitrary domains specified by an expert-written DSL. In order to efficiently support arbitrary domains, we developed a synthesizer that combines two key ideas. First, we use a novel iterative synthesis technique inspired by test-driven development (TDD) [12, 22], giving our methodology the name of *test-driven synthesis* (TDS). Second, we have advanced the state-of-the-art in DSL-based synthesis as our iterative synthesis only shines when paired with a algorithm that can efficiently improve the program from a previous step.

Iterative synthesis Taking inspiration from the test-driven development (TDD) [12, 22] concept of iterating on a program that always works for the tests written so far, the behavior of functions in LaSy is constrained by a *sequence* of increasingly sophisticated examples like those that might be written by a programmer authoring a function using the TDD programming methodology or demonstrating the functionality via examples to another human.

Prior PBE systems take as input a *set* of *representative* examples from which a program is synthesized (they may be consumed one at a time, but their order is not prioritized). It is desirable that the user provides representative examples; otherwise the system may end up over-fitting on a simple example, and would fail to find a common program that works for all examples [17]. On the other hand, when describing a task people can easily provide a *sequence* of examples of increasing complexity—for example, in a human-readable explanation [26] or in a machine-readable sequence of tests for TDD. The presentation as a sequence allows for the problem of learning the task to be broken down into a series of sub-problems of learning slightly more detail about the task after seeing each new example—this is one of the key insights of this paper. While this bears some similarity to genetic programming [23], our system does not rely on any concept of a function almost or approximately matching an example. By working in a series of steps where after each step the result is a program that is correct for a subset of the input space, our system is able to more effectively narrow down the search space than prior work.

* Work done during two interships at Microsoft Research

[1] Language for Synthesis, pronounced "lazy"

PLDI '14, June 09–11, 2014, Edinburgh, United Kingdom.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2784-8/14/06.
http://dx.doi.org/10.1145/2594291.2594297

DSL-based synthesis The underlying synthesis technique, which we refer to as DBS for DSL-based synthesis, is parameterized by a DSL allowing it to be specialized to different domains with minimal effort. Similar to how a parser generator is given a language definition to make a parser for that language, our general-purpose synthesizer is given a DSL definition to make a synthesizer for that DSL. A DSL primarily consists of a context-free grammar over DSL-defined functions. Additionally, the DSL may use synthesis strategies we provide for learning constructs amenable to special reasoning like conditionals and certain forms of loops. Experts may provide further extensions, but that is out of scope for this paper as we focus on the strengths of DBS and our test-driven synthesis methodology without the complication of additional user-provided synthesis strategies.

As it is part of an iterative synthesis procedure, DBS is given a previous program that satisfies some prefix of the examples and searches for a modification that makes it satisfy all of the examples. These modifications replace some subexpression of the previous program with a new expression. The generation of replacement subexpressions is done by component-based synthesis [15]: DBS considers the semantically distinct expressions of the DSL, building them up starting with the atoms in the DSL and the previous program's subexpressions. In our DSL-based approach, DBS uses the grammar of the DSL as opposed to just the types of the expressions to decide what expressions to build, which significantly narrows the search space.

Contributions

1. The problem statement of solving a PBE problem given a sequence of increasingly sophisticated input/output examples for each function to be synthesized, reified in our LaSy language (§3).
2. Our novel *test-driven synthesis* methodology for solving such synthesis problems combines iterative synthesis (§4) and DSL-based synthesis (§5) to be able to synthesize programs with non-trivial control flow including conditionals, loops, and recursion in an arbitrary DSL.
3. Being domain-agnostic, our technique has myriad applications, including significant implications for end-user programming; we summarize a few possiblities with concrete examples in §2.
4. We run experiments demonstrating the effectiveness of each of our key ideas and showing the overall effectiveness of our algorithm on important applications including comparing against state-of-the-art synthesizers in each domain and against the general purpose synthesizer Sketch [30] (§6). As a source for more difficult problems for our system, particularly for long sequences of examples, we had it play the Pex4Fun programming game [32] (§6.1.4).

2. Motivating examples

We present examples demonstrating the breadth of programs LaSy is able to synthesize to highlight its domain-agnostic capabilities. §6 goes into more detail on evaluating the performance and design decisions of our synthesizer.

2.1 Automating TDD

To demonstrate iteratively building up a program, we use the example of greedy word wrap, whose use as a TDD practice problem in [22] was the original inspiration for our methodology. Word wrap inserts newlines into a string so that no line exceeds a given maximum line length; the greedy version inserts newlines as late as possible instead of trying to even out the line lengths.

Fig. 1 shows part of a LaSy program for implementing greedy word wrap, showing only 1 or 2 of the up to 6 examples under

```
language strings;
function string WordWrap(string text, int length);

// Single word doesn't wrap.
require WordWrap("Word", 4) == "Word";
// Two words wrap...
require WordWrap("Extremely longWords", 14) ==
"Extremely\nlongWords";
// ... when longer than line.
require WordWrap("How are", 76) == "How are";
// Wrap as late as possible...
require WordWrap("How are you?", 9) == "How are\nyou?";
// ... but no later.
require WordWrap("Hello, how are you today?", 14) ==
"Hello, how are\nyou today?";
// Wrap in middle of word.
require WordWrap("Abcdef", 5) == "Abcde\nf";
require WordWrap("ThisIsAVeryLongWord a", 15) ==
"ThisIsAVeryLong\nWord a";
// Wrap multiple times (using recursion).
require WordWrap("How are you?", 4) == "How\nare\nyou?";
// Complicated test to ensure program is correct.
require WordWrap("This is a longer test sentence. a bc",
7) == "This is\na\nlonger\ntest\nsentenc\ne. a bc";
```

Figure 1. Abbreviated LaSy program for greedy word wrap showing a representative subset of the 24 test cases

each comment for space reasons. The LaSy program begins with a declaration saying to use the "strings" DSL followed by the function signature for word wrap. Most of the program is the list of examples that dictate how the function should behave.

The examples are interspersed with comments that explain how this sequence iteratively builds up a solution with intermediate steps where it is easy to describe what subset of the inputs the program is correct for. Exactly where to break the line is refined over several examples: first it is just at the space, then at the space in a string longer than the maximum line length, then at the last space in a long string, then at the last space before (or at) the maximum line length, only to be refined further toward the end to include inserting breaks in the middle of words that do not fit on a single line. Also, our synthesizer is able to generalize from inserting a single line break to inserting any number of line breaks via recursion.

The high number of examples indicates this is a fairly sophisticated program for PBE; it should be noted that full test coverage for TDD takes a similar number of examples [22]. We include word wrap to show LaSy is capable of scaling up to that many examples.

2.2 End-user data transformations

Enabling (hundreds of millions of) end-users to construct small scripts for data manipulation is a very important topic [1, 6]. LaSy can be used to synthesize useful scripts in this domain that are beyond the capabilities of leading prior work in this area [6].

String transformations Consider the task of converting bibliography entries (represented as strings) from one format to another. Fig. 2 shows an example of a LaSy program for converting between two such formats. This program is structured using helper functions for rewriting the authors list and for mapping venue abbreviations to the full venue names used in the target format. Unlike ConvertName and ConvertList, VenueFullName does not do any computation (the full name for "POPL 2013" cannot be inferred from the full name for "PLDI 2012" without a web search), so it is declared as a lookup, meaning the function will just store the list of input/output examples and look up any inputs in that list to find the corresponding output.

```
language strings;
function string ConvertBib(string oldFormat);
function string ConvertName(string fullName);
function string ConvertList(string oldFormat);
lookup string VenueFullName(string abbr);
require ConvertName("John Smith") == "Smith, J.";
require ConvertName("Ann Miller") == "Miller, A.";
require ConvertList("John Smith") == "Smith, J.";
require ConvertList("Donna Jones, John Smith, Ann
Miller") == "Jones, D.; Smith, J.; Miller, A.";
require VenueFullName("PLDI 2012") == "The 33rd ACM
SIGPLAN conference on Programming Language Design and
Implementation, Beijing, June, 2012";
require ConvertBib("Reagents: Expressing and Composing
Fine-grained Concurrency, PLDI 2012, Aaron Turon")
== "Reagents: Expressing and Composing Fine-grained
Concurrency\nTuron, A.\nThe 33rd ACM SIGPLAN conference
on Programming Language Design and Implementation,
Beijing, June, 2012";
require VenueFullName("CACM 2012") == "Communications of
the ACM, 2012";
require ConvertBib("Spreadsheet Data Manipulation using
Examples, CACM 2012, Sumit Gulwani, William Harris,
Rishabh Singh") == "Spreadsheet Data Manipulation
using Examples\nGulwani, S.; Harris, W.; Singh, R.\n
Communications of the ACM, 2012";
```

Figure 2. LaSy program for converting bibliography entries

```
oldXml = "<doc><div id="ch1"> <p name="a1">1st
Alinea.</p> <p name="a1.1">Zomaar ertussen.</p> <p
name="a2">2nd Alinea.</p> <p name="a3">3rd Alinea.</p>
</div> <div id="ch2"> <p name="a1">First Para.</p> <p
name="a2">Second Para.</p> <p name="a2.1">Something
added here.</p> <p name="a3">Third Para.</p>
</div></doc>";
language xml;
function XDocument ToTable(XDocument oldXml);
function XElement BuildRow(XDocument oldXml,
    string rowName);
require BuildRow(oldXml, "a1.1") == "<tr><td>Zomaar
ertussen.</td><td/></tr>";
require ToTable(oldXml) == "<table>
<tr><td>1st Alinea.</td><td>First Para.</td></tr>
<tr><td>Zomaar ertussen.</td><td/></tr>
<tr><td>2nd Alinea.</td><td>Second Para.</td></tr>
<tr><td/><td>Something added here.</td></tr> <tr><td>3rd
Alinea.</td><td>Third Para.</td></tr> </table>";
```

Figure 3. LaSy program for converting set of XML lists to a table

XML transformations We show two LaSy programs for XML transformation tasks found on help forums. Fig. 3 takes a set of `<div>`s containing named paragraphs and arranges the data as a table with a column for each `<div>` and a row for each name, so data is lined up in a row if the same name appears in multiple `<div>`s. This transformation requires a helper which describes how a table row is built from a paragraph name. Both functions are simple enough that they require only a single example. Fig. 4 assigns class attributes to paragraphs without them according to the class of the nearest previous paragraph (if present), and is implemented by the synthesizer using a loop.

3. Language

In addition to the LaSy programming by example language demonstrated in the previous section, which is explained in more detail in §3.1, we also discuss the language that experts use to define DSLs for use in LaSy in §3.2.

```
language XML;
function XDocument AddClasses(XDocument oldXml);
require AddClasses("<doc> <p>1</p> </doc>") == "<doc>
<p>1</p> </doc>";
require AddClasses("<doc> <p>1</p> <p class='a'>2</p>
<p>3</p> <p>4</p> <p class='b'>5</p> <p>6</p> <p
class='c'>7</p> </doc>")
== "<doc> <p>1</p> <p class='a'>2</p> <p class='a'>3</p>
<p class='a'>4</p> <p class='b'>5</p> <p class='b'>6</p>
<p class='c'>7</p> </doc>";
```

Figure 4. LaSy program for adding class attributes

$\mathbf{P}$	$::=$	`language` $\mathbf{I}$`;` $\mathbf{F}*$ $\mathbf{E}*$	(Program)
$\mathbf{F}$	$::=$	`function` $\mathbf{t}$ $\mathbf{f}$`(`$(\mathbf{t}\ \mathbf{x},)*$`);`	(Function declaration)
		$\mid$ `lookup` $\mathbf{t}$ $\mathbf{f}$`(`$(\mathbf{t}\ \mathbf{x},)*$`);`	(Lookup declaration)
$\mathbf{E}$	$::=$	`require` $\mathbf{f}$`(`$(\mathbf{V},)*$`)` `==` $\mathbf{V}$`;`	(Example)
$\mathbf{V}$	$::=$	any constant expression	(Value)
$\mathbf{t}$	$::=$	$\mathbf{I} \mid \mathbf{I}$`<`$(\mathbf{t},)*$`>`	(type name)
$\mathbf{f}$	$::=$	$\mathbf{I}$	(function name)
$\mathbf{x}$	$::=$	$\mathbf{I}$	(variable name)
$\mathbf{I}$	$::=$	any valid identifier	(Identifer)

Figure 5. The LaSy language.

3.1 LaSy programming by example language

Fig. 5 gives the syntax of LaSy. Note that LaSy relies on a base language, C# in our implementation, to provide basic types, functions, and values, and therefore the precise syntax for values and identifiers is omitted: type references are C# type references and values are C# expressions.

Programs in LaSy consist of a set of function declarations and a sequence of examples.

Language All LaSy programs begin with a reference to the DSL being synthesized over. The language is defined beforehand by an expert as described below in §3.2.

Functions The function declarations list the functions to be synthesized. Each function has a name and type signature.

Examples and semantics Each example is a function call with literals given for its arguments and its required return value. A function `f` is considered to satisfy an example `f`$(V_1,\ldots)$`==`V_R if whenever `f` is called with arguments that are structurally equivalent (`.Equals()` in C#) to $V_1,\ldots$, it returns a value that is structurally equivalent to V_r.

Specifically, the examples are an ordered list dictating a sequence of program synthesis operations wherein the function the example references is modified to satisfy the example without violating any previous example. At the beginning of the synthesis of a LaSy program, all functions are empty (and therefore satisfy no examples). The synthesis process is described in detail in §4.

The semantics of LaSy are naturally quite weak: the only guarantee is that if the synthesis succeeds, then the examples will be satisfied. The synthesizer is heavily biased toward smaller programs with fewer conditionals which tend to be more generalizable, but does not guarantee it will find the smallest program satisfying all of the examples. While the synthesizer implementation should act in a manner predictable enough that the user can trust its programs to be reasonable, ultimately only the user can determine if the synthesized program actually fulfills the user's intended purpose.

The result of the synthesizer is C# code which can be compiled and used in any .NET program, including another LaSy program.

```
language strings;
assembly flashfill.dll class lasy.FlashFill;
start P;

P ::= __CONDITIONAL(b, e);
b ::= ||(d, ..., d);
d ::= &&(π, ..., π);
π ::= m | !(m);
m ::= Match(v, r, k) | Match(f, r, k) | <(i, i);
i ::= Length(v) | GetPosition(v, p) | j;
j ::= _PARAM;
e ::= Concatenate(f, ..., f)
    | SplitAndMerge(v, s, s, λf:e);
f ::= ConstStr(s) | SubStr(v,p,p) | Loop(λw:e)
    | SubStr(f,p,p) | Trim(f)
    | _LASY_FN(f) | _RECURSE(f, j);
s ::= _CONSTANT;
p ::= Pos(r,r,c) | CPos(k)
    | CPos(c) | CPos(j) | RelPos(p,r,c);
c ::= k | k*w+k;
k ::= _CONSTANT;
r ::= TokenSeq(T,...,T) | ϵ;
v ::= _PARAM;

rewrite &&(π_0, π_1) ==> &&(π_1, π_0);
rewrite ||(d_0, d_0) ==> d_0;
rewrite 0*w_0+k_0 ==> k_0;
rewrite Trim(Trim(f_0)) ==> f_0;
```

Figure 6. The extended FlashFill DSL; grammar rules not present in the original DSL are in **bold**.

3.2 DSL definition language

An example DSL definition is given in Fig. 6. The DSL gives a grammar which specifies what programs are possible as well as what the semantics of those programs are. Additionally, the DSL optionally provides a few different kinds of hints to the synthesizer that allow the synthesizer to take advantage of expert knowledge of the semantics.

Grammar The DSL is given as a context-free grammar. Each line defines an option for a non-terminal given on the left of the `::=`. For most rules, the right side gives a DSL-defined function and a list of non-terminals for the arguments to that function. Functions are .NET functions defined in the class specified at the top of the file. The semantics of the DSL must be functional; that is, all functions called must be pure. Note that loops can be handled in a pure way by using lambdas. In general a while loop can be written using the function `WhileLoop(condition, body, final) = state => condition(state) ? WhileLoop(condition, body, final)(body(state)) : final(state)`.

Rules other than DSL-defined functions are written in all-caps starting with an underscore to distinguish them. These are used for a few different purposes. First, some are items that are not functions like constants, lambda variables, and lambda abstraction. Some reference items depend on the LaSy program: `_PARAM` corresponds to any parameter of the correct type and `_LASY_FN` allows for a call to another LaSy function. Those starting with a double underscore are for functions with specialized synthesis strategies. `__CONDITIONAL(b, e)` means that some number of `if...else if...else` branches are allowed where the conditions match the non-terminal **b** and the branches match the non-terminal **e**; this could be a DSL-defined function except for the fact that the synthesizer is aware of the semantics of conditionals and has specialized logic for learning them described in §5.2. Similarly, while not used in the example, `__FOR(i)` and `__FOREACH(arr)` have associated synthesis strategies for certain forms of loops which are described in §5.3.

Constant value generation The DSL may include code to decide what constant values may appear in the program which may depend on the examples (not shown in the figure). The simplest logic a DSL could use is that any values in the examples may be used as constants in the program. Other DSLs may have cases like including only regular expressions from a preset list that match one of the inputs or, when synthesizing XML, extracting the names of the tags and attributes in the outputs.

Rewrite rules The **`rewrite`** rules allow the DSL designer to express algebraic identities in their language to help prune the search space of programs with identical semantics.

Algorithm 1: TDS(S, $\mathcal{L}$)

```
input  : sequence of examples $S$, DSL specification $\mathcal{L}$
output : a program $P$ that satisfies $S$ or FAILURE
$e \leftarrow$ all grammar rules in $\mathcal{L}$;
$P_0 \leftarrow \bot$;
failuresInARow $\leftarrow 0$;
foreach $i \leftarrow 0, \ldots, |S|-1$ do
    if $P_i(input(S_i)) = output(S_i)$ then
        $P_{i+1} \leftarrow P_i$;
        failuresInARow $\leftarrow 0$;
    else
        contexts $\leftarrow \emptyset$, exprs $\leftarrow e \cup$ parameters of $P_i$;
        foreach subexpression $s$ of $P_i$ do
            Add $\lambda expr.P_i[s/expr]$ to contexts;
            Add $s$ to exprs;
        foreach branch $B$ of $P_i$ do
            foreach subexpression $s$ of $B$ do
                Add $\lambda expr.B[s/expr]$ to contexts;
        $P_{i+1} \leftarrow$ DBS(contexts, $(S_0, \ldots, S_i)$, exprs, $\mathcal{L}$,
                num_branch($P_i$)+failuresInARow);
        if $P_{i+1}$ is TIMEOUT then
            $P_{i+1} \leftarrow P_i$;
            failuresInARow $\leftarrow$ failuresInARow $+ 1$;
        else
            failuresInARow $\leftarrow 0$;
if failuresInARow $= 0$ then
    return $P_{|S|}$;
else
    return FAILURE;
```

4. Test-Driven Synthesis

The Test-Driven Synthesis methodology synthesizes a program by considering a sequence of examples in order and building up iteratively more complicated programs. We will describe the methodology for a LaSy program with one function, but it easily generalizes to multiple functions. §4.1 describes the algorithm in detail. §4.2 details how the iterative nature of the algorithm works. §4.3 discusses the importance of the order of examples.

4.1 Algorithm

The Test-Driven Synthesis algorithm (TDS in Algorithm 1) synthesizes a program P given a sequence of examples S and a set of base components. By "program" we mean a single function with a specified set of input parameters and return type. By "example" we mean a set of input values for those parameters and the correct output value. By "component" we mean any of the set of expressions known to the synthesizer which are used as the building blocks for the synthesized program; the base components are the functions referenced by the DSL in the LaSy program but components may also be partially filled-in function calls or larger DSL expressions.

In the spirit of TDD, we build P up, a little at a time, to allow synthesis of larger programs. P_i satisfies the first i examples; its successor program P_{i+1} is built by the DSL-based synthesis (DBS) algorithm using the first $i + 1$ examples along with information from P_i. The previous program P_i is used in three ways:

1. Its subexpressions are added to the component set.
2. *Contexts* to synthesize in are formed by removing each subexpression of P_i one at a time.
3. The number of branches may only exceed the number of branches in P_i if `failuresInARow` > 0. New conditionals are allowed only after failures in order to avoid overfitting to the examples by creating a separate branch for each one.

EXAMPLE 1 (Walkthrough of TDS). *We will use the DSL* $\boldsymbol{C} ::=$ *CharAt*$(\boldsymbol{S},\boldsymbol{N})|$*ToUpper*$(\boldsymbol{C})$, $\boldsymbol{S} ::=$ *Word*$(\boldsymbol{S},\boldsymbol{N})|$*_PARAM,* $\boldsymbol{N} ::= 0|1$ *where* `Word(s, n)` *selects the* n^{th} *word from the string* `s` *and _PARAM is any function parameter and* e *is the set of all grammar rules in that DSL to demonstrate synthesizing the function* `f(a)⇒ToUpper(CharAt(Word(a, 1), 0))`*:*

i=0: $S_0 = (a =$ *"Sam Smith"*, $RET =$ *'S')*. *exprs* $= e \cup \{a\}$ *(the whole language plus the parameter* `a`*) and contexts* $= \{\circ\}$ *(the set containing just the trivial context) because the previous program* $P_0 = \bot$*, so there are no subexpressions to remove to build contexts out of. The smallest program to compute* `'S'` *is to select the first character of* a*, and therefore* $P_1 =$ *f(a)* $\Rightarrow$ *CharAt*$(a, 0)$.

i=1: $S_1 = (a =$ *"Amy Smith"*, $RET =$ *'S')*. *contexts* $= \{\circ,$ *CharAt*$(\circ, 0),$ *CharAt*$(a, \circ)\}$ *because* P_1 *has two subexpressions that can be removed. exprs now also contains the expression CharAt*$(a, 0)$*. The simplest program consistent with both examples selects the second word of* `a` *instead of* `a` *itself, so DBS will generate Word*$(a, 1)$ *to select the second word and plug it into the second context to generate* $P_2 =$ *f(a)* $\Rightarrow$ *CharAt*$($*Word*$(a, 1), 0)$.

i=2: $S_2 = (a =$ *"jane doe"*, $RET =$ *'D')*. *contexts*$= \{\circ,$ *CharAt*$(\circ, 0),$ *CharAt*$($*Word*$(\circ, 1), 0),$ *CharAt*$($*Word*$(a, \circ), 0)\}$. *exprs* $= e \cup \{a,$ *Word*$(a, 1),$ *CharAt*$($*Word*$(a, 1))\}$*. Notably, CharAt*$(a, 0)$ *does not appear in exprs despite it appearing in exprs for the* $i = 1$ *step because it is not a subexpression of* P_2*. It is important that such temporary diversions are forgotten so time is not wasted on them in later steps. DBS will output* $P_3 =$ `f(a)⇒ToUpper(CharAt(Word(a, 1), 0))` *which takes only a single step because it is the application of ToUpper to* P_2 *which appears in exprs.*

We now discuss a few details of the algorithm to clarify the description and justify some design choices.

Relation between *TDS* ***and*** *DBS* DBS is described later in §5. We separate TDS from DBS both to show explicitly how the previous program P_i is used when constructing the next program P_{i+1} and to highlight the two key novel ideas in our approach:

1. TDS encapsulates the new idea of treating the examples as a sequence and using that fact to iteratively build up P by way of a series of programs which are correct for a subset of the input space defined by a prefix of the examples.
2. DBS encapsulates the parameterization by a DSL which allows for the flexibility of the algorithm.

TDS runs DBS repeatedly, each time giving it the next example from S along with expressions and contexts from the program synthesized in the previous iteration. In other words, it iteratively synthesizes P_k for each $k \leq |S|$ where P_k is synthesized using $S_0, S_1, \ldots, S_{k-1}$ and the previous program P_{k-1}. In this formulation, P_0 is the empty program $\bot$, or `throw new NotImplementedException();` in C#.

State As described, the only state kept between iterations is the program P_i and the failure count. DBS does not maintain state, and `contexts` and `exprs` depend only on P_i. Additionally, DBS is passed all of the examples up to S_i, not just S_i. One could imagine a more general problem definition where arbitrary (or at least more) state could be kept between invocations of DBS, but in our experience this tended to be more harmful than helpful: **preserving state essentially corresponds to not forgetting about failed attempts.**

No lookahead Although we have formulated the problem as giving TDS the sequence of tests, notice it does not look beyond test S_i to generate P_{i+1}. Hence in an interactive setting the user could look at P_{i+1} or its output when choosing S_{i+1}.

4.2 Contexts and subexpressions

The intuition for the strategy of replacing subexpressions is that the program generated so far is doing the correct computation for some subset of the input space and is overspecialized to that subset. In the example above, after the $i = 0$ step, we had the program that returns the first character of the string instead of the first character of the second word of the string. That program was overspecialized to inputs where the first and second word start with same letter. Selecting the first letter was the right computation but on the wrong input, so filling in the context CharAt$(\circ, 0)$ with the right input gave the desired program.

Each context represents a hypothesis about which part of the program is correct and correspondingly that the expression removed is overspecialized. Note that the expression appears in the set of components, so if a small change is sufficient, the effort to build it in previous iterations will not be wasted. Also, one such hypothesis is always that the entire program is wrong and should be replaced entirely.

Contexts are made out of each branch as well as the entire program in order to better support building new conditional structures (§5.2) using parts of one or more of the existing branches.

This theory does not limit contexts to a single hole, but, empirically, doing so keeps the number of contexts manageable and seems to be sufficient in practice. Also, it allows the algorithm to prune away locations based on whether they are reached when executing a failing example: modifications elsewhere could not possibly affect whether such examples are handled correctly. If we allowed multiple modifications, the choice of modification points would have to be changed after any modification affecting control flow.

4.3 Example order

[22] observes that in TDD the test case order can affect the ability of the programmer to produce a program through small code changes. Similarly, our algorithm may fail to synthesize a program if not given examples in a good order—after all, one of our key insights is that the ordering of examples is a useful input to the synthesizer.

As a "good" order is defined as being one that results in synthesizing a program satisfying the specification the user has in mind, it is unclear how to define a "good" order without referencing the final synthesized program. Needless to say, such a definition cannot be directly used to guide the generation of a sequence of examples. This is unsatisfying, but we provide some intuition on what such orders look like and §6.2 gives evidence that our synthesizer is robust to small variations in the order of examples. We thus remark that a human could learn to produce such an ordering just like a human can learn to produce TDD test cases in an order that easily results in a correct program. Fundamentally, this is analogous to the issue of how a human should generate a concise but sufficient set of black-box tests for a program. Many guidelines exist, but there is no precise methodology. Nonetheless, black-box testing is successful.

Once the user has covered the entire specification they had in mind, they have produced a suite of simple test cases for the algorithm they are synthesizing. As in TDD, to confirm that they have in fact synthesized the correct procedure, the user should write a few larger, more comprehensive tests of the procedure.

5. DSL-Based Synthesis

The DSL-Based Synthesis algorithm (DBS in Algorithm 2) is the part of TDS that actually generates new programs. DBS takes as input a set of examples S, a set of contexts C which generated DSL expressions are plugged into to form synthesized programs matching $\mathcal{L}$, a set of expressions e, a DSL definition $\mathcal{L}$, and a maximum number of branches m. It outputs a new program P' that satisfies all examples in S or TIMEOUT if it is unable to do so.

The algorithm is built on five key concepts:

1. New programs are not generated directly; instead expressions are generated and plugged into contexts provided as hints to narrow the search space. This is used by TDS to indirectly provide the previous program as a hint (§4.2).
2. New expressions are formed from all compositions of expressions according to the DSL $\mathcal{L}$. To produce all smaller expressions before generating larger ones, DBS runs as a series of iterations, where, in each iteration, only expressions from previous iterations are composed into new expressions. §5.1 discusses generation of new expressions (and important optimizations).
3. A new branching structure will be synthesized if no generated program satisfies all examples in S and $m > 1$. Only programs containing at most m branches will be synthesized in order to avoid over-specializing to the examples. Synthesis of conditionals is discussed in detail in §5.2.
4. If the algorithm times out before a solution is found, it will return a special failure value TIMEOUT. In TDS, this case increments m allowing for more branches in the next run of DBS.
5. The search space can be reduced even further using specialized strategies for some functions. We demonstrate this by describing strategies we defined for a couple common loop forms in §5.3.

5.1 Choosing new expressions

New expressions to use in the contexts are generated by component-based synthesis [14]. In component-based synthesis, a set of components (expressions and methods) are provided as input and iteratively combined to produce expressions in order of increasing size until an expression is generated that matches the specification. In our case, the "specification" is the examples. As opposed to previous component-based synthesis work, the generation of new expressions is guided by a DSL $\mathcal{L}$ and instead of testing the expressions against the specification, they are used to fill in contexts producing larger programs which are then tested.

In our system, all components are expressions marked with which non-terminal in the grammar defined them. Methods are represented as curried functions. The synthesizer generates new expressions by taking one curried function and applying it to an expression marked with the correct non-terminal. Each iteration of the synthesizer does so for every valid combination of previously generated expressions in order to generate programs of increasing size. Representing methods as anonymous functions also simplifies

Algorithm 2: DBS(C, S, e, $\mathcal{L}$, m)

```
input  : set of contexts $C$, set of examples $S$, set of
         expressions $e$ to build new expressions from, DSL
         specification $\mathcal{L}$, maximum number of branches $m$
output : a program $P'$ that satisfies $S$ or TIMEOUT
/* Try generates one or more programs and if
   one satisfies $S$, DBS returns it.               */
1 Try loop strategies in a separate thread (§5.3);
2 allExprs ← $e$;
3 while not timed out do
4     foreach $c \leftarrow C$ do
5         foreach $expr \leftarrow$ allExprs do
6             Try $c(expr)$;
7     Try conditional solutions up to $m$ branches (§5.2);
8     allExprs ← generate new expressions (§5.1);
9 return TIMEOUT;
```

handling methods that themselves take functions as arguments, which are common in higher-order functions like map and fold.

As the number of components generated after k iterations is exponential with the base being the number of grammar rules (i.e., functions and constants in the DSL) in the worst case, a DSL that is too large will cause DBS to run out of time or memory before finding a solution. In practice, around 40–50 grammar rules seems to be the limit for DBS, but it depends greatly on the structure of the DSL. An earlier version of DBS without the optimizations described below could not handle more than around 20–30 grammar rules. Further optimizations to better prune the search space could possibly allow for even larger DSLs.

Optimizations

Minimizing the number of generated expressions is important for performance. Redundant expressions are eliminated in two ways: the first is syntactic and hence it is fast and always valid, while the second is semantic and valid only when an expression does not take on multiple values in a single execution (e.g., if the program is recursive).

Syntactic All expressions constructed are rewritten into canonical forms according to the **rewrite** rules in the DSL and duplicates are discarded. For example, x+y and y+x are written differently but can be rewritten into the same form so one will be discarded. DBS will only accept sets of **rewrite** rules which are acyclic (once commutativity and other easily broken cycles are removed) to ensure there is a canonical form. Related to this, constant folding is applied where possible, so, for example, 2*5 and 5+5 would both be constant folded to 10, further reducing the search space.

Semantic The vast majority of the time, an expression takes on only a single value for each example input. In other words, the expression is equivalent to a lookup table from the example being executed to its value on that example. Only expressions with distinct values are interesting, so, for example x*x and 2+x would be considered identical if the only example inputs were x = 2 and x = -1. This is similar to the redundant expression elimination in version space algebras [18]. The exceptions are if the expression is part of a recursive program or lambda expression, in which case this optimization is not used.

5.2 Conditionals

So far we have not considered synthesizing programs containing conditionals, which are of course necessary for most programs. We consider first synthesizing programs where a single cascading sequence of `if...else if...else` expressions occur at the top-level of the function body, with each branch not containing conditionals. Then the goal is to have as few branches in the one top-level conditional as possible. The problem is to partition the examples into which-branch-handles-them to achieve this goal.

For every program p DBS tries, the set of examples it handles correctly is recorded and called $T(p)$. If $T(p) = S$ (all examples handled), p is a correct solution and can be returned. Otherwise, each set of programs Q (where $|Q| \leq m$) whose union of handled examples $\bigcup_{p \in Q} T(p)$ equals S is a candidate for a solution with appropriate conditionals. To be a solution, Q also needs guards that lead examples to a branch that is valid for them; to simplify this, whenever a boolean expression g is generated, the set of examples it returns true for, $B(g)$, is recorded. The sets Q are considered in order of increasing size, so if there are multiple solutions, the one with the fewest branches will be chosen.

If the conditional does not appear at the top-level, then it must appear as the argument to some function. To handle this case, we note that if every branch of the conditional generated as described already happens to contain a call to a function f with different arguments, then it could be rewritten such that the call to f occurs outside of the conditional if that is allowed by the DSL. In that case, we can say that all of the branches match the context $f(\circ)$.

In the algorithm, for each non-terminal the DSL allows for conditionals at, each program p is put into zero or more buckets labeled with the context that non-terminal appears in. For example, if the argument of f may be a conditional and $p = f(f(x))$ then p would be put in the buckets for $f(\circ)$ and $f(f(\circ))$. Then the same algorithm as above is run for each bucket with the conditionals being rewritten to appear inside the context. Inserting multiple conditionals just involves following this logic multiple times.

5.3 Loops

The primary way DBS handles loops is to simply not do anything special at all: recursion and calling higher-order functions like `map` and `fold` are handled by the algorithm as described so far. As described in §3.2, a general `WhileLoop` higher-order function can be used to express arbitrary loops that DBS may synthesize like any other DSL-defined function. On the other hand, the use of loops in code often corresponds to patterns in the input/output examples. This section discusses two such common patterns we have written strategies for; experts designing DSLs may additionally define their own strategies for other forms of loops.

These can be used in a DSL via the `__FOREACH(E)` or `__FOR(E)` rules where **E** is the non-terminal for the body of the loop.

Foreach The "foreach" loop strategy's hypothesis is that there is a 1-to-1 correspondence between an input array and an output array. Assuming that hypothesis, the examples can be split into one example for each element where `i` is the index, `current` is the element at that index, and `acc` is the array of outputs for previous indexes: $(\texttt{in} = \{3, 5, 4\}, \texttt{RET} = \{9, 25, 16\})$ would become the examples $(\texttt{in} = \{3, 5, 4\}, \texttt{i} = 0, \texttt{current} = 3, \texttt{acc} = \{\}, \texttt{RET} = 9)$, $(\texttt{in} = \{3, 5, 4\}, \texttt{i} = 1, \texttt{current} = 5, \texttt{acc} = \{3\}, \texttt{RET} = 25)$, and $(\texttt{in} = \{3, 5, 4\}, \texttt{i} = 2, \texttt{current} = 4, \texttt{acc} = \{9, 25\}, \texttt{RET} = 16)$. Those examples could be used to synthesize the loop body `current*current` using TDS. The strategy includes the boilerplate code to take the loop body `current*current` and output a `foreach` loop over the input array.

That example is overly simple as such a computation could easily be captured by a `map`. However, loop strategies also allow for loops that are not as easily expressed with higher-order functions. For example, the loop bodies examples could also include the values computed so far: $(\texttt{in} = \{5, 2, 3\}, \texttt{RET} = \{5, 7, 10\})$ would become $(\texttt{in} = \{5, 2, 3\}, \texttt{i} = 0, \texttt{current} = 5, \texttt{acc} = \{\}, \texttt{RET} = 5)$, $(\texttt{in} = \{5, 2, 3\}, \texttt{i} = 1, \texttt{current} = 2, \texttt{acc} = \{5\}, \texttt{RET} = 7)$, and $(\texttt{in} = \{5, 2, 3\}, \texttt{i} = 2, \texttt{current} = 3, \texttt{acc} = \{5, 7\}, \texttt{RET} = 10)$. Then the synthesized loop body would be `acc.Length > 0 ? current + acc.Last() : current`, which could be rewritten into a loop computing the cumulative sum of `in`.

For Patterns may also show up across examples. For instance, given the examples $(\texttt{in} = 0, \texttt{RET} = 0)$, $(\texttt{in} = 1, \texttt{RET} = 1)$, $(\texttt{in} = 2, \texttt{RET} = 3)$, $(\texttt{in} = 3, \texttt{RET} = 6)$ we can see, looking across examples, that for each input value the result should be the result for the previous input value plus the new input (which is an indirect way of saying "sum the numbers up to `in`"). In terms of loop strategies, the hypothesis is that pairs of examples where the input `in` differ by one correspond to adjacent loop iterations so by combining those pairs we can get examples for the loop body where `i` is current value of the loop iterator and `acc` is the return value of the $\texttt{i} - 1$ iteration: $(\texttt{i} = 1, \texttt{acc} = 0, \texttt{RET} = 1)$, $(\texttt{i} = 2, \texttt{acc} = 1, \texttt{RET} = 3)$, and $(\texttt{i} = 3, \texttt{acc} = 3, \texttt{RET} = 6)$. Then TDS will give `i + acc` for the loop body. The loop strategy will identify that $(\texttt{in} = 0, \texttt{RET} = 0)$ indicates that the loop iterator should start at 0 and the accumulator should start at 0 and produce a `for` loop `for(int i = 1; i <= in; i++) acc = i + acc;`.

Other strategies Different loop strategies can give different information like including the index in a `foreach` or giving `acc` corresponding to going in reverse order. Furthermore, the concept of spliting up arrays by element to find patterns can also be applied to splitting strings (by length or delimiters), XML nodes, or whatever other structured data may be in the target domain.

5.4 Conditionals and loops, a general theory

DSL definitions contain rules with and without specialized strategies. Most rules, including all DSL-defined functions, do not have specialized strategies so expressions using those rules are built using the default strategy of searching through the semantically distinct expressions (using the example inputs to decide which expressions are distinguishable). On other other hand, we have defined strategies for conditionals and loops that use the example outputs as well as the inputs to power more directed approaches to learning those constructs. While we referenced the output values directly in the explanation of the strategies for loops, the discussion of conditionals only referenced them indirectly by keeping track of which examples a program was correct for.

The strategies for conditionals and loops should be considered as just different instances of the same concept. While conditionals merely select a subset of the examples for each branch, loops do larger rewrites of the examples used in the recursive calls to the synthesizer. Theoretically, a DSL designer could include other strategies like inverses of DSL-defined functions or a polynomial solver for synthesizing arithmetic. We presently omit such functionality because we believe it places undue burden on the DSL designer but intend to investigate it in future work.

6. Evaluation

Our evaluation demonstrates that TDS is sufficiently powerful and general to synthesize non-trivial—albeit small—programs in multiple domains from small sequences of real world examples obtained from help forums. We also compare TDS to to Sketch [30], the present state-of-the-art in domain-agnostic program synthesis, and to state-of-the-art specialized synthesizers where applicable.

Furthermore, we explored how sensitive our iterative synthesis technique actually is to the precise ordering of examples, showing that example order is significant to speed or ability to synthesize on a non-trivial proportion of the examples, especially on larger programs. We also validated some of our design decisions by selectively disabling parts of our algorithm.

All experiments were run on a machine with a quad core Intel Xeon W3520 2.66GHz processor and 6GB of RAM.

§6.1 describes the benchmarks used in our experiment and our success in synthesizing them and compares TDS to prior specialized synthesizers where applicable. §6.2 investigates the effect of example ordering to our synthesizer's performance. §6.3 breaks down the usefulness of the different parts of our algorithm. §6.4 evaluates performance for our synthesizer and validates our timeout choice.

6.1 Benchmarks

We evaluate our technique in four domains: §6.1.1 compares our technique to a state-of-the-art specialized programming by example system for string transformations, §6.1.2 compares our technique to a start-of-the-art specialized programming by example system for table transformations, §6.1.3 discusses using our system for the novel domain of XML transformations, while finally §6.1.4 shows our technique is able to automate coding in TDD for introductory programming problems.

For the first three, the example sequences used and output of our synthesizer can be found at `https://homes.cs.washington.edu/~perelman/publications/pldi14-tds.zip`; the last section's programs are not public.

6.1.1 String transformations

String transformation programs take as input one or more strings and output a string constructed from the input using mainly substring and concatenation operations.

Strings are a natural format for input/output examples that often appear in real-world end-user programming tasks as recent work by Gulwani et al. [6] has shown: their work became the FlashFill feature in Excel 2013 [1]. Unlike FlashFill, TDS does not use careful reasoning about the domain of strings, but it is still able to quickly synthesize many of the same examples as well as some similar programs that the prior work (i.e., FlashFill) cannot synthesize.

Benchmarks To compare against FlashFill, we first defined exactly the FlashFill DSL in our DSL definition language and ran our synthesizer on the examples which appear in [6] to confirm we could synthesize them. Then we made a few modifications to the DSL which are shown in Fig. 6 to make it more general; specifically, we allowed nested substring operations, substring indexes dependent on the loop variable, and calls to other LaSy functions.

In addition to WordWrap and the examples from [6] we wrote test case sequences for 7 simple real world string manipulation examples outside of the scope of FlashFill but handled by our extended DSL including selecting the two digit year from a date (requires nested substrings), reversing a string (requires substring indexes dependent on the loop variable), and bibliography examples like the one in Fig. 2 (requires a user-defined lookup).

Results Each of the 15 example sequences contains 1–8 examples except for word wrap which uses 24 examples. 5 of the examples can be synthesized in under a second. 6 take more than 1 second but under 5 seconds, while the other 4 finish in under 25 seconds. FlashFill synthesizes all of the examples it can handle in well under a second. Simply by specifying the DSL, our domain-agnostic synthesis technique nears the performance of a state-of-the-art specialized synthesis technique while maintaining the ability to generate more complicated control flow structures.

We coded all examples using the corresponding DSLs in Sketch and none of them completed within 10 minutes.

6.1.2 Table transformations

Table transformations convert spreadsheet tables between different formats by rearranging and copying a table's cells.

Benchmarks [11] gives a DSL and synthesis algorithm for these transformations along with a collection of benchmarks the authors found on online help forums. We defined their DSL for our synthesizer and ran it on their benchmarks.

For additional benchmarks not handled by [11] we added more predicates to the grammar to allow it to handle a wider range of real world normalization scenarios. For example, our extended grammar can support converting various non-standard spreadsheets with subheaders into normalized relational tables.

Results Each of the 8 benchmarks uses 1–6 examples. TDS synthesizes most of them in under 10 seconds; 2 take 30 seconds and one takes a full minute.

[11] says Sketch was unable to synthesize their benchmarks so we did not attempt to run the benchmarks using Sketch.

6.1.3 XML transformations

XML transformations involve some combination of modifying an XML tree structure and tag attributes and string transformations on the content of the XML nodes.

Benchmarks We selected 10 different real world examples from online help forums and constructed a DSL able to express the operations necessary to synthesize programs for all of them. Two of the examples appear in §2.2.

One transformation involved putting a tag around every word, even when words had tags within them, which was easiest to express treating the words as strings instead of as XML. Making the string and XML DSLs work together required simply putting the functions to convert between the two in the DSL. This kind of cross-domain computation shows the strength of our domain-agnostic approach.

Results Most of the benchmarks used a single example while the rest used no more than 3. TDS synthesizes all but two of the benchmarks in under 10 seconds and the remaining two in under 20 seconds.

We also implemented the DSL and benchmarks in Sketch, which was unable to synthesize any of them within 10 minutes.

6.1.4 Pex4Fun programming game

Motivation Although we believe our end-user programming scenarios are compelling, we wanted to test our synthesizer in a scenario closer to the TDD programming style it was inspired by and get a source for some more challenging functions to synthesize. To that end, we had our synthesizer play the Pex4Fun [32] programming game where a player is challenged to implement an unknown function given only a few examples. Each time the player thinks they have a solution, the Pex [31] test generation tool uses dynamic symbolic execution to compare the player's code to a secret reference solution and generates a distinguishing input if the player's code does not match the specification. Pex provides the test for the test step of the TDD while the player is performing the programming step without full knowledge of the specification they are coding for. This is a more "pure" form of TDD as the player is not biased in their coding by knowing the specification, giving a closer parallel to our synthesizer which does not have knowledge of the specification of the function it is synthesizing.

Experiment description We use a single DSL with a set of 40 simple `string` and `int` functions which may be combined in any type-safe way to show that while a carefully constructed DSL can be given to our synthesizer to make it perform especially well in a given domain, it can still successfully synthesize programs using a less specialized DSL capable of describing a wider range of programs. Note that our DSL was written without looking at the

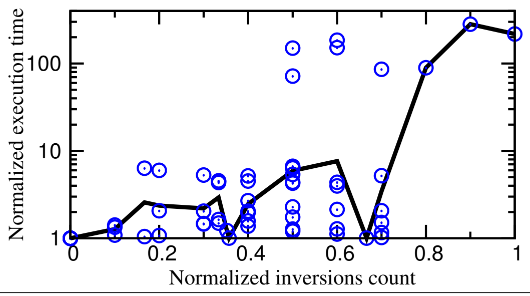

Figure 7. Norm. time (1=opt) for many reorderings of examples (dist. measured in inversions, norma. so 1=reverse); the line shows the geo. mean of the data points for each norm. inversion count.

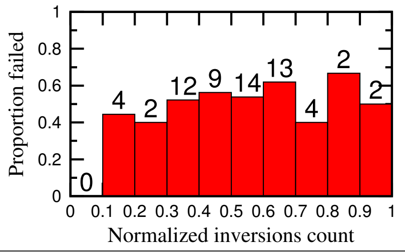

Figure 8. Prop. of reorderings TDS failed on by norm. inversion count. Numbers above bars are absolute # of failed reorderings.

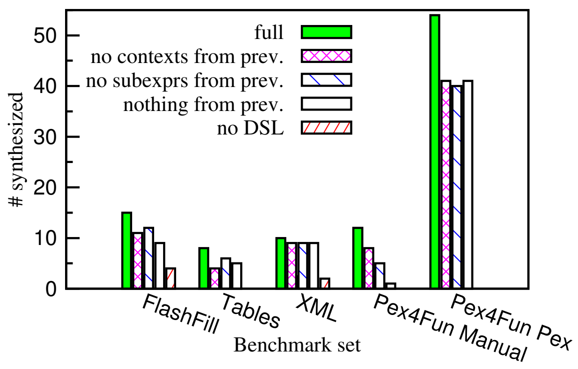

Figure 9. # synthesized by benchmark set and features enabled in algorithm. "full" is the full algorithm, contexts and subexpressions from previous program together form the information TDS uses.

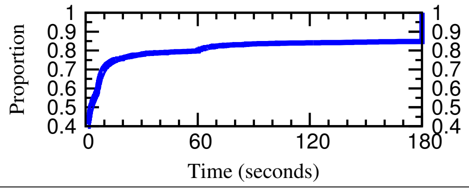

Figure 10. Execution times of all DBS runs.

Pex4Fun puzzles and therefore ended up missing some functions necessary for some puzzles like bitwise operations.

For each puzzle in the Pex4Fun data, we had our algorithm play Pex4Fun for a maximum of 7 iterations, after which the synthesizer was considered to have failed if it still had not produced a solution. This may seem like too few iterations, but it is more calls to Pex than most users used for any of the puzzles.

For some of the puzzles, using Pex to generate test cases failed to generate a solution despite manual inspection determining that the puzzles were well within the capabilities of the synthesizer and the DSL used. In such cases, a sequence of test cases was generated manually to synthesize solutions to those puzzles.

Benchmarks The puzzles our synthesizer could synthesize included, among many others, factorial, swapping elements of an array, and summing numbers in a string delimited by delimiter specified on the first line of the string and some more trivial examples like concatenating the first and last element of a string array.

The remaining unsynthesized puzzles either involved looping structures not covered by our strategies (e.g., count the number of steps of the $3n + 1$ problem[2] needed to reach 1), components not in our component set (e.g. compute bitwise or), or arithmetic expressions too large to construct using component-based synthesis (e.g. compute the value of a specific cubic polynomial).

Results We ran our experiment across 172 randomly selected puzzles from Pex4Fun. The synthesizer found solutions for 72 of those. For 60 of those, the test cases generated by Pex were sufficient, but for another 12 the test cases had to be written manually.

6.2 Example ordering

We hypothesized that example ordering is useful and TDS is robust to small variations in example order. For the vast majority of our benchmarks, the number of examples is small and their order is unimportant; the use of contexts and subexpressions from the previous program is important (when not synthesizing from a single example) as is shown in §6.3 by disabling them, but the specific order of the examples is not. On the other hand, the 12 Pex4Fun puzzles which required manually written example sequences were difficult enough for TDS that the ordering of examples was in fact valuable to the algorithm. To give an idea of how important the ordering actually was, we ran TDS on randomly reordered copies of those example sequences.

Fig. 7 shows the timing results for the example sequences that were successfully synthesized. Each circle is one example sequence, and the line shows geometric mean of circles.

The x-coordinate measures how far from the optimal example sequence it was where 0 is the optimal sequence and 1 is the reverse of the optimal sequence. Specifically, the value is the number of inversions[3] between the two sequences divided by the maximum possible number of inversions ($\frac{n(n-1)}{2}$ for a sequence of length n). The y-coordinate is the time it took to synthesize the solution using the random sequence measured in the time it took to synthesize using the optimal sequence. Note that the y-axis is a log scale.

Fig. 8 shows for how many of the reorderings the program was not successfully synthesized. The x-axis is the same; each bar corresponds to a range of normalized inversion counts. The bar heights are the proportion of sequences for which a program could not be synthesized, and the numbers above the bars are the absolute counts. There are more examples toward the middle as the sequences were selected uniformly at random, and there are more possible sequences in the middle.

The charts show two important properties of TDS. **First, it does in fact depend on example ordering**: large changes to the example ordering make it take much longer to find a solution or fail to find a solution at all. **Second, it is robust to small changes in the example ordering**: for normalized inversion counts less than 0.3, fewer than half the reorderings failed and those that succeeded took on average less than three times as long as the optimal ordering.

Those 12 examples show the worst case for our synthesizer. For the other 60 Pex4Fun puzzles with test cases sequences from Pex,

[2] https://en.wikipedia.org/wiki/Collatz_conjecture

[3] # of example pairs that have different order in the two lists

51 of them were also successfully synthesized with those test cases in reverse order. For the rest of the examples, nearly all could be synthesized with the examples sequence in reverse order, at worst slowed down by a factor of 3. This indicates that the sensitivity to test case ordering is affected by how complicated the program being synthesized is.

6.3 Significance of various parts of algorithm

In order to evaluate the usefulness of the different parts of our algorithm, we ran our benchmarks with parts of it disabled. Fig. 9 shows how many of the benchmarks were synthesized under each limited version of the algorithm.

Iterative synthesis The iterative synthesis strategy employed by TDS is implemented by passing contexts and subexpressions from the previous program to DBS. We disabled these two pieces of information individually and together. The Pex4Fun and table transformation benchmarks were most affected by the removal of features from TDS due to working with larger programs. Notably, we see that **either the subexpressions or contexts alone is helpful, but when combined they are significantly more powerful.**

DSL-based synthesis We also disabled the use of the DSL when generating components in DBS, so it instead would be limited only by the types of expressions. There are no "no DSL" bars for the Pex4Fun benchmarks because the Pex4Fun DSL already only used the types, so that configuration is identical to the "full" configuration. Many of the other programs were synthesized from just a single example, so the weakening of TDS did not have a large effect, but this success was achieved due to the power DBS gained from the DSL as can be seen from the fact that very few of the end-user benchmarks could be synthesized without access to the DSL.

6.4 Performance

Fig. 10 shows a CDF of all execution times of all of the DBS runs used in our experiments. **This chart shows that DBS is quite efficient with a median running time of approximately 2 seconds and running in under 10 seconds around 75% of the time.**

Timeout Throughout the experiments, we used a 3 minute timeout. Only very rarely in our experiments did DBS ever run for anywhere near 3 minutes without timing out. There is a visible bump around 60–70 seconds after which the line is almost flat, indicating that it is very unlikely that giving DBS a small amount of additional time would have made any noticeable difference in our results. This is further verified by ad-hoc experience that without a timeout, DBS runs for over 30 minutes without a result or runs out of memory.

7. Related Work

Programming by example Programming by example (PBE) [7], or inductive programming [16], is a subfield of program synthesis covering many different techniques where a program is incompletely specified by examples (inputs paired with explicit outputs or a quality function for results). In all of the prior work, all of the examples are given at once, although much of it uses some variant of genetic programming [23] where programs are built and iteratively mutated toward a solution or CEGIS [30] where new programs are constructed until a solver fails to provide a counterexample, similar to how our synthesizer was used with Pex in the Pex4Fun experiment. A key idea in the recent prior work is reducing the search space using version space algebras [6, 8, 11, 18, 28, 29], which we avoid in order to maintain generality as they must be constructed for a given domain.

TDS is most similar to prior work on component-based synthesis and genetic programming.

Component-based synthesis Component-based synthesis is a family of techniques that perform program synthesis by starting with a set of "components"—that is, expressions and functions—and considering actual programs constructed from combining these components (as opposed to the version space algebra approach where the actual program is merely an artifact that can be recovered after the main synthesis algorithm is complete). Component-based synthesis has been successfully applied to a wide variety of domains including bit-vector algorithms [9], string transformations [24], peephole optimizations [5], type convertors [21, 27], deobfuscation [13], and geometry constructions [10].

The actual search is performed in different ways dictated by the information required be known about the components. For example, Gulwani et al. [9] used an SMT solver because their components are standard bitwise arithmetic operators, so they have easily expressible logical specifications. Also, this is the prior work we found with the largest program synthesized by component-based synthesis at 16 lines of code, which took around an hour to synthesize. In comparison, our algorithm can synthesize programs of up to 20 lines of code within 400 seconds consisting of arbitrary user-defined functions.

The prior work most similar to our DBS is by Katayama [15]. Like our algorithm, any function can be a component and it maintains an explicit list of the components generated so far and uses them when generating components on the next step. It finds and prunes redundant components by evaluating every component on a subset of the examples and only keeping components that evaluate to different values on some example. This is similar to the pruning done by DBS except that DBS uses all examples seen so far.

Genetic programming Genetic programming [23] synthesizes programs by performing a genetic search which involves using previous candidate programs to produce new candidate programs. The primary difference between our work and genetic programming is that genetic programming is directed by a quality function that tells how well a given program performs at the desired task and relies on that quality function being well-behaved while our search is only given a boolean failure or success on each test case.

ADATE [25] takes as input a set of test inputs and quality functions for their outputs and performs a genetic search where programs are mutated and only programs that improve or maintain the sum of the quality function values are kept. This formulation means that ADATE has to choose which test case to improve next, unlike in our system where the first k test cases must pass before the synthesizer considers test case $k + 1$.

Template-based synthesis Program sketching [30] is a form of program synthesis where the programmer writes a *sketch*, i.e., a partial program with holes, and provides a specification the solution must satisfy. LaSy can be seen as a less precise sketch with a DSL instead of partial bodies: in fact, we ported our LaSy programs to Sketch in order to do a comparison for our evaluation. However, the search strategy is very different: our synthesizer fills in the holes using component-based synthesis (as opposed to using SAT/SMT solvers) and checks against input/output examples (instead of a more complete specification).

Syntax-guided synthesis Our DSL-based approach is similar to the syntax-guided synthesis idea proposed in [2], but is more flexible by allowing for a DSL that uses arbitrary .NET functions. That work [2] only presents simple prototype synthesizers which lack the power of TDS. SyGuS's use of constraints instead of examples makes a direct comparison of synthesizer technologies difficult.

Rosette [33] is an extension to the Racket programming language which allows easy access to a solver for applications including synthesis over DSLs embedded in Racket. While the programmer experience is much sleeker than with SyGuS or LaSy, the synthesis engine suffers similar limitations to Sketch: it cannot efficiently handle types that don't map to an SMT solver.

Automated program repair In automated program repair [19, 34, 35], many passing and failing test cases are given along with a buggy human-written program to repair. This is a different task because the automated program repair systems are not expected to add new functionality, only fix existing functionality. As a result, the existing code can be effectively used to guide many repairs.

In angelic debugging [3], expressions that are likely to be the right place to make a change are identified. For each expression, it determines if it is a possible repair point by taking a set of passing and failing test cases and checking if for each test case, there is some alternative value for that expression that makes the test case pass. Angelic debugging does not attempt to synthesize new expressions; it only identifies the possible locations for fixes. Like our algorithm, angelic debugging assumes that a program can be repaired by changing only a single expression. In fact, angelic debugging could be used as a preprocessing step in our algorithm to prune the choices for modification points or determine that a simple modification is unlikely to work.

8. Conclusions and Future Work

Our synthesis technique advances the state-of-the-art in component-based synthesis which has been restricted to synthesis of straight-line code fragments over a given set of components. Our overall test-driven synthesis methodology enables synthesis of programs containing conditionals and loops over a given set of components. As future work, we intend to explore including information about function inverses in the DSL, expanding the range of control flow structures supported by our synthesizer, use the synthesizer to generate feedback for the Pex4Fun game and introductory programming assignments, create an interface for performing end-user programming tasks with LaSy, and explore other applications for TDS utilizing its incremental nature including updating sythesized code as a specification changes or fixing code from another synthesizer that generates approximate or incomplete solutions.

9. Acknowledgements

This work was funded in part by two internships at Microsoft Research. We thank the anonymous reviewers for their valuable feedback. We thank Nikolai Tillmann and Peli de Halleux of Microsoft Research's Pex4Fun team for access to the Pex4Fun data.

References

[1] `http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html`.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[3] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. ICSE, 2011.

[4] A. Cypher, D. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. Myers, and A. Turransky. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.

[5] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. PLDI, 1992.

[6] S. Gulwani. Automating string processing in spreadsheets using input-output examples. POPL, 2011.

[7] S. Gulwani. Synthesis from examples: Interaction models and algorithms. SYNASC, 2012.

[8] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8), Aug. 2012.

[9] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. PLDI, 2011.

[10] S. Gulwani, V. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. PLDI, 2011.

[11] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. PLDI, 2011.

[12] D. Janzen and H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9), sept. 2005.

[13] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. ICSE, 2010.

[14] S. Katayama. Systematic search for lambda expressions. TFP, 2005.

[15] S. Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI*. 2008.

[16] E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In U. Schmid, E. Kitzelmann, and R. Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*. 2010.

[17] T. Lau et al. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, 2008.

[18] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1), 2003.

[19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1), Jan. 2012.

[20] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[21] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. PLDI, 2005.

[22] R. Martin. The transformation priority premise. `http://blog.8thlight.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html`, 2010.

[23] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. ONeill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[24] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. ICML, 2013.

[25] R. Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1), Mar. 1995.

[26] R. Panigrahy and L. Zhang. The mind grows circuits. *CoRR*, abs/1203.0088, 2012.

[27] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. PLDI. ACM, 2012.

[28] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8), 2012.

[29] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.

[30] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[31] N. Tillmann and J. De Halleux. Pex: white box test generation for .NET. TAP, 2008.

[32] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Pex4Fun: Teaching and learning computer science via social gaming. CSEE&T, 2012.

[33] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152. ACM, 2013.

[34] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5), May 2010.

[35] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. ICSE, 2009.