FlashFill++: Scaling Programming by Example by Cutting to the Chase

JOSÉ CAMBRONERO*, Microsoft, USA
SUMIT GULWANI*, Microsoft, USA
VU LE*, Microsoft, USA
DANIEL PERELMAN*, Microsoft, USA
ARJUN RADHAKRISHNA*, Microsoft, USA
CLINT SIMON*, Microsoft, USA
ASHISH TIWARI*, Microsoft, USA

Programming-by-Examples (PBE) involves synthesizing an *intended* program from a small set of user-provided input-output examples. A key PBE strategy has been to restrict the search to a carefully designed *small* domain-specific language (DSL) with *effectively-invertible* (EI) operators at the top and *effectively-enumerable* (EE) operators at the bottom. This facilitates an effective combination of top-down synthesis strategy (which backpropagates outputs over various paths in the DSL using inverse functions) with a bottom-up synthesis strategy (which propagates inputs over various paths in the DSL). We address the problem of scaling synthesis to large DSLs with several non-EI/EE operators. This is motivated by the need to support a richer class of transformations and the need for readable code generation. We propose a novel solution strategy that relies on propagating fewer values and over fewer paths.

Our first key idea is that of *cut functions* that prune the set of values being propagated by using knowledge of the sub-DSL on the other side. Cuts can be designed to preserve completeness of synthesis; however, DSL designers may use incomplete cuts to have finer control over the kind of programs synthesized. In either case, cuts make search feasible for non-EI/EE operators and efficient for deep DSLs. Our second key idea is that of *guarded DSLs* that allow a *precedence* on DSL operators, which dynamically controls exploration of various paths in the DSL. This makes search efficient over grammars with large fanouts without losing recall. It also makes ranking simpler yet more effective in learning an intended program from very few examples. Both cuts and precedence provide a mechanism to the DSL designer to restrict search to a reasonable, and possibly incomplete, space of programs.

Using cuts and gDSLs, we have built FlashFill++, an industrial-strength PBE engine for performing rich string transformations, including datetime and number manipulations. The FlashFill++ gDSL is designed to enable readable code generation in different target languages including Excel's formula language, PowerFx, and Python. We show FlashFill++ is more expressive, more performant, and generates better quality code than comparable existing PBE systems. FlashFill++ is being deployed in several mass-market products ranging from spreadsheet software to notebooks and business intelligence applications, each with millions of users.

CCS Concepts: • **Software and its engineering → Programming by example**; *Domain specific languages.*

*Authors in alphabetic order

Authors' addresses: José Cambronero, Microsoft, USA, jcambronero@microsoft.com; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Vu Le, Microsoft, USA, levu@microsoft.com; Daniel Perelman, Microsoft, USA, danpere@microsoft.com; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Clint Simon, Microsoft, USA, clint.simon@microsoft.com; Ashish Tiwari, Microsoft, USA, astiwar@microsoft.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/1-ART33
https://doi.org/10.1145/3571226

Additional Key Words and Phrases: programming by example, domain-specific languages, string transformations

ACM Reference Format:
José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (January 2023), 30 pages. https://doi.org/10.1145/3571226

1 INTRODUCTION

Programming-by-examples (PBE) has seen tremendous interest and progress in the last decade [Gulwani et al. 2017]. A variety of approaches have been proposed targeting various applications. Starting from purely symbolic techniques, the field has explored neural [Devlin et al. 2017] and neurosymbolic approaches [Chaudhuri et al. 2021; Kalyan et al. 2018; Rahmani et al. 2021; Verbruggen et al. 2021]. In this paper, we present novel symbolic techniques to improve the scalability of PBE systems.

PBE applications range from enabling non-experts to author programs for spreadsheet data manipulation [Gulwani et al. 2012] or application creation in a low-code/no-code setting [Lukes et al. 2021], to improving productivity of data scientists for data wrangling tasks [Le and Gulwani 2014; Miltner et al. 2018] and even automating professional developers' repeated edits [Pan et al. 2021; Rolim et al. 2017]. A flagship application for PBE is that of string transformations, for instance, converting `'Alan Turing'` to `'turing, alan'`—such tasks are very common and are well described by examples [Gulwani 2011]. We focus on such string transformations, though our technical contributions are more generally applicable to any grammar-based synthesis setting.

Most PBE engines work by defining a program search space, and then employing some strategy to search over it. The program space is often defined by a domain-specific language (DSL), which fixes a finite set of operators and all the different ways they can be composed to create programs. A key challenge in PBE is scaling the search to very large program spaces. DSL designers have to build DSLs that are expressive enough to be useful, yet small enough to keep the program search space small. This tension in DSL design has hindered broader applications of program synthesis. Users implicitly advocate for larger DSLs as they want synthesizers to produce programs that are closer to the ones they would manually write, i.e., ones that use a large variety of functions that are available in general purpose programming languages. On the other hand, these larger DSLs (a) make the search space large and the synthesis slow, and (b) more importantly, allow the large number of functions to be combined in unintuitive ways to produce undesirable programs. Program synthesis research has mainly focused on completeness, i.e., ensuring that we find a program when one exists, and insisting on completeness for large DSLs exacerbates these problems. We introduce two new mechanisms, *cuts* and *precedence*, by which DSL designers can control the program search space even as the DSL itself grows in size. This not only eases the job of the DSL designer, but also enables them to build synthesizers based on very expressive DSLs.

Challenges. Let us say we are given an input-output example: a tuple of input values and one output value. There are two commonly used search strategies to find programs that would generate the output value using the input values: bottom-up and top-down.

Bottom-up (BU) search starts with the inputs and generates *all possible values that can be computed from the inputs* using all possible (partial) programs in the search space. It does so by applying the *executable semantics functions* of the DSL operators. Thus, information flows from the inputs, and all computed intermediate results are completely oblivious to the output. The BU strategy is effective only when the sets of values generated in each intermediate stage remains small. This happens when there are only a small number of leaf constants and each operator is *effectively-enumerable* (EE) – namely, it has a small arity and many-to-one semantics, thus having a small dynamic fan-out.

Top-down (TD) search starts with the output and applies the inverse semantics of the operators, so-called witness functions, to generate intermediate *values that can generate the output value.* Thus, information flows from the output value, and at every stage, the intermediate values are computed solely based on the output and are completely oblivious to the input values. The TD strategy is effective only when these intermediate sets are small. This happens only when every operator is *effectively invertible* (EI) – namely, it allows for effective (inverse) computation of various inputs that can yield a given output.

If the DSL has both non-EE and non-EI operators, then neither bottom-up nor top-down strategies are effective. Recently, it was observed that one could scale synthesis to larger DSLs by combining the two strategies [Lee 2021]. If there is a partition of the DSL such that the sub-DSL closer to the start symbol has EI operators, and the rest contains EE operators, then the two strategies can be combined to yield a *meet-in-the-middle* strategy at that cut [Lee 2021]. However, this new strategy still has only a limited form of information flow between the inputs and the output. In fact, the DSLs used in Duet [Lee 2021] are relatively small, albeit larger than those in FlashFill and FlashMeta [Gulwani 2011; Polozov and Gulwani 2015]. These latter systems are based on a TD strategy over very small DSLs.

An alternate way to scale PBE synthesis is based on abstraction and refinement types [Feng et al. 2017; Guo et al. 2020; Polikarpova et al. 2016; Wang et al. 2017]. The idea behind abstraction is that instead of computing the *exact* set of values that can be generated (in either top-down or bottom-up strategy), we compute *overapproximations* of the sets of values. This approach works when high quality abstractions can be quickly generated. Typically, it is still "one-sided" – either the inputs flow to intermediate values or the output value flows backwards to intermediate values. Furthermore, abstractions of compositions of operators are computed by composing the abstractions of operators, which loses accuracy as the composition depth grows. We overcome some of these shortcomings in our work; however, abstraction-based approaches are inherently complementary.[1]

Our Contribution. In this paper, we present two novel techniques - cuts and precedence - to effectively address the scalability challenges of PBE synthesis. The executable semantics functions (that are the basis of BU) and the witness functions (that are basis of TD) are defined to allow information to flow in one direction. We introduce *cuts* that prune values generated by witness functions guided by the values that sub-DSLs could possibly compute on the inputs. This concept inherently builds in bi-directional information flow in its definition, and in fact, generalizes the semantics functions and witness functions. A DSL designer can author a *cut function* based on their intuition of the form of values that can be computed at a nonterminal using the inputs, and then restricting them to those that would be relevant for the output. Unlike abstractions, cuts are not computed compositionally. They are provided for whole sub-DSLs; thus, they avoid information loss accumulated by composing lossy abstractions. This is similar to how *accelerations* avoid information loss in program analysis by capturing the effect (composition or transitive closure) of multiple state transitions by a single "meta transition" [Finkel 1987; Karp and Miller 1969].

A top-down strategy would get stuck at a non-EI operator. However, a cut function for an argument of that non-EI operator can help unblock TD synthesis. As a special case, a cut for that argument can be generated using bottom-up enumeration, in which case we get the meet-in-the-middle strategy [Lee 2021]. However, cuts may be generated by other means based on the DSL designer's insight. In general, we get a novel search strategy, *middle-out synthesis*, which uses cuts

[1] In this paper, we focus exclusively on synthesis approaches based on concrete values: the specification is a concrete IO example, and the semantics functions (and the inverse semantics) are given on concrete values (and not abstract values or refinement types). More specifically, we are in the context of version-space algebra (VSA) driven synthesis, and hence the terms top-down and bottom-up are always used in that context.

to reduce the original PBE problem over a large DSL (with potentially non-EI and non-EE operators) into simpler PBE problems over (smaller depth) sub-DSLs with only EI or only EE operators.

Our second key idea is to introduce precedence over operators in the grammar of domain-specific languages. Precedence is a natural concept in grammars and arises naturally when the DSL designer wants to prefer certain operators over others. We show that if the precedence is a series-parallel partial order, then it can be encoded as an ordering on grammar rules to create a *guarded DSL* (gDSL), and program search can be performed directly on the gDSL without compromising soundness or completeness, while gaining efficiency. The ordering on rules in a gDSL is interpreted as a mandate to explore a certain branch only when higher-ordered branches have failed to return a result. This has two major advantages. First, it makes the search more scalable by dynamically using different underapproximations of the DSL to be explored. Second, it makes ranking simpler to write for DSL designers because the precedence already builds in a default ranking over programs.

Cuts and precedence provide DSL designers two new mechanisms to control the program search space, beyond what they get through designing DSLs. Our contributions include:

- A new algorithmic approach for PBE (*middle-out synthesis*) that leverages a novel *cut rule* to speed up synthesis over large DSLs and to handle non-EI and non-EE operators.
- A new formalism of guarded DSLs that supports operator precedence, and an extension of our synthesis approach to gDSLs that scales synthesis to large DSLs and gives ranking based on path orderings [Dershowitz and Jouannaud 1990] for free.
- A new and expressive system FlashFill++ for string transformations that supports datetime & number manipulations and is designed for readable code generation.
- An extensive comparison of FlashFill++ with existing state-of-the-art PBE systems for string transformations (FlashFill [Gulwani 2011], SmartFill [Chen et al. 2021a], and Duet [Lee 2021]) that shows improvements in expressiveness, learning performance, and code readability.

2 OVERVIEW

2.1 New Challenges in PBE for String Transformations

We first discuss some challenges faced by the current generation of PBE tools for string transformations. These challenges were compiled in collaboration with two key industrial deployers of PBE: the Microsoft Excel and the Microsoft PowerBI teams. To compile these challenges, we interviewed several product managers in these teams, interacted with both expert and novice users of the FlashFill feature, and analyzed online help posts.

Generating Readable Code. Consider the task of transforming the input pair `("David Walker", "623179")` to the output string `"D-6231#walker"`. Any string processing library would contain many *redundant* methods for extracting `"Walker"` from `"David Walker"`. For example, in Python, we could use the `split` method to accomplish the task. Alternatively, we could use the `find` method along with string slicing, or use regular expressions.

In contrast to the design of string processing libraries, the prevailing wisdom in DSL design for synthesis has been to work with a minimal number of operators [Gulwani 2016]. For example, FlashFill [Gulwani 2011] and the more recent Duet [Lee 2021] DSLs contain only 3 and 5 functions that directly operate on strings, respectively. Smaller DSLs lead to smaller program search spaces, yielding better synthesis performance and effective ranking [Polozov and Gulwani 2015]. Following this minimalism to an extreme can lead to a DSL whose programs translate to very unnatural and unreadable programs in target languages like Python (see Figure 1) or PowerFx (see Figure 2).

One straightforward approach to readability is writing a good translator from the synthesis DSL to the target language. However, if the semantic gap between the DSL and the target languages' operators is large, then "readable translation" itself becomes a new and nontrivial synthesis problem.

```python
# Python program generated by FlashFill
import regex
def transformation_text_program(input1, input2):
    computed_value_83 = regex.search(r"\p{Lu}+", input1).group(0)
    index1_83 = regex.search(r"[-.\p{Lu}\p{Ll}0-9]+", input2).start()
    computed_value_0_83 = input2[index1_83:(len(input2) + -2)]
    kth_match_83 = list(regex.finditer(r"[-.\p{Lu}\p{Ll}0-9]+", input1))[-1]
    computed_value_1_83 = kth_match_83.group(0).lower()
    return computed_value_83+"-"+computed_value_0_83+"#"+computed_value_1_83
```

```python
# Python program generated by FlashFill++
def formula(i1, i2):
    s1 = i1[:1]
    s2 = i2[:4]
    s3 = i1.split(" ")[1].lower()
    return s1 + "-" + s2 + "#" + s3
```

```python
# Python program generated by FlashFill++ and renamed by Codex
def formula(name, number):
    first_initial = name[:1]
    number_prefix = number[:4]
    last_name = name.split(" ")[1].lower()
    return first_initial + "-" + number_prefix + "#" + last_name
```

Fig. 1. The python programs generated by FlashFill and FlashFill++ for the task of transforming `("David Walker", "623179")` to `"D-6231#walker"`. FlashFill++'s program is much more readable and its readability is further improved by renaming variables using a pretrained large language model, as shown.

Our insight is that to effectively generate readable code the DSLs should not be designed with the single-minded goal of efficient learning, but also pay heed to the target languages.

While generating readable code is challenging, the need is sorely felt in industrial PBE tools—users are more likely to trust and use PBE tools if they produce idiomatic, readable code. Quoting one study participant in [Drosos et al. 2020]: *"don't know what is going on there, so I don't know if I can trust it if I want to extend it to other tasks. I saw my examples were correctly transformed, but because the code is hard to read, I would not be able to trust what it is doing".* The lack of readable code is one of the primary challenges preventing a broader adoption of PBE technologies.

Multiple Target Languages. The need for readable code generation is compounded by the proliferation of different target languages, each with their own set of operations; see Figure 3. These target languages range across standard programming languages (e.g., Python, R), individual libraries (e.g., Pandas, PySpark), data query languages (e.g., SQL), and custom application-specific languages (e.g., Google Sheets & Excel formula languages, PowerBI's M language). Apart from the obvious benefit, multiple target support can also help with learning: seeing the same program in multiple languages helps with cross-language knowledge transfer [Shrestha et al. 2018].

Date-Time and Numeric Transformations. Most string PBE technologies cannot natively handle date-time and numeric operations efficiently, leading to situations like transforming `'jan'` to `'Janember'` given the input-output example `'nov'` $\mapsto$ `'November'`. Duet [Lee 2021] does allow for limited numeric operators, but still lacks support for important data-processing operations such as rounding and bucketing. According to the Microsoft Excel team, date-time and numeric operations (of the kind shown in Figure 3) are among the most requested FlashFill features. However, as illustrated in Section 2.2, these operations are not amenable to standard synthesis techniques.

```
# PowerFx formula generated by FlashFill
Concatenate(
  Mid(Left(input1, Match(input1, "\p{Lu}+").StartMatch
                   + Len(Match(input1, "\p{Lu}+").FullMatch) - 1),
      Match(input1, "\p{Lu}+").StartMatch),
Concatenate("-",
Concatenate(Mid(Left(input2,Len(input2)-2), Match(input2,"[0-9]+").StartMatch),
Concatenate("#",
  Lower(Mid(
    Left(input1,
      First(LastN(MatchAll(input1, "[\p{Lu}\p{Ll}]+"), 1)).StartMatch
      + Len(First(LastN(MatchAll(input1, "[\p{Lu}\p{Ll}]+"), 1)).FullMatch)-1),
    Last(MatchAll(input1, "[\p{Lu}\p{Ll}]+")).StartMatch)))))
```

```
# PowerFx formula generated by FlashFill++
Left(input1, 1) & "-" & Left(input2, 4) & "#"
  & Lower(Last(FirstN(Split(input1, " "), 2)).Result)
```

Fig. 2. PowerFx formulas generated by FlashFill and FlashFill++ to transform ("David Walker", "623179") into "D-6231#walker". The latter is much more readable than the former.

Performing operations over datetimes and numbers allows our system to handle use cases like the one detailed in Fig. 4(a), which presents 911 call records that need to be transformed. Each call log, shown in the *Input* column, contains an (optional) address, the township, the call date and time, followed by possible annotations indicating the specific 911 station that addressed the call. Let us suppose that a data scientist wants to extract the date (2015-12-11) and time (13:34:52) from each log, and map it to the corresponding weekday (Fri) and the 3-hour window (12PM - 3PM), as shown in the *Output* column. Performing this transformation requires string processing to extract candidate dates and times, parsing these substrings into appropriate datatypes, performing type-specific transformations on the extracted values, and then formatting them into an appropriate output string value. This is beyond the capabilities of current synthesizers. Our system can synthesize the intended program (shown in Fig. 4(b)) from just the first example. This program is readable and also serves educational value (e.g. teaching the API of the popular datetime Python library).

2.2 Overview of FlashFill++

We now show how our novel techniques address the various challenges from subsection 2.1.

Extended Domain-Specific Language. The main strength of FlashFill++ compared to previous systems is its expanded DSL containing over 40 operators, including 25 for just strings and the rest for datetime and numbers, such as for rounding and bucketing; see Figure 7. Contrast this with the numbers 3 and 5 mentioned previously for FlashFill and Duet.

This extended DSL supports more expressive *and* more readable programs. Contrast the code generated by FlashFill++ in Fig. 1 and 2 to that generated by FlashFill to see the clear difference an extended DSL makes. However, expanding the DSL comes with its own set of challenges: (a) Given the larger search space, standard synthesis techniques fall short on efficiency. (b) The larger search space also complicates ranking—the problem of picking the best (or intended) program among all the ones consistent with the examples. (c) Handling numeric and date-time operators requires new synthesis techniques. Next, we discuss some novel strategies to address these challenges.

Cuts and Middle-Out Synthesis. Our new DSL contains several non-EI operators (required for number and datetime operations) that inhibit use of a top-down synthesis strategy across those operators. Furthermore, bottom-up synthesis is not feasible for the sub-DSLs below those operators

Round to last day of month: 2/5/2020 $\Longrightarrow$ 2/29/2020

```
from datetime import datetime
from dateutil.relativedelta import *

def formula(i1):
  month_start = datetime(i1.year,i1.month,1)
  month_end = month_start
     + relativedelta(months=1)
  return month_end - relativedelta(days=1)
```

```
EOMONTH(A1)
```

```
With({monthStart:
  Date(Year(i1), Month(i1), 1)
},
 DateAdd(
   DateAdd(
     monthStart, 1, ""Months""),
   -1, ""Days""
))
```

Round to start of quarter: 2/5/2020 $\Longrightarrow$ 1/1/2020

```
from datetime import datetime

def formula(i1):
  quarter = (i1.month - 1) // 3 + 1
  return datetime(i1.year,3*quarter-2,1)
```

```
EOMONTH(
  DATE(YEAR(A1),
    ROUNDUP(MONTH(A1)/3,
                0)*3,1),
  0)
```

```
With({quarter:
  RoundUp(Month(i1) / 3, 0)
},
 Date(Year(i1),quarter*3-2,1)
    + Time(0, 0, 0))
```

Days since start of year: 2/5/2029 $\Longrightarrow$ 36

```
def formula(i1):
  return i1.timetuple().tm_yday
```

```
A1 - DATE(YEAR(A1), 1, 1) + 1
```

```
DateDiff(
  Date(Year(i1),1,1),i1) + 1
```

Create year-quarter string: 4/5/1983 $\Longrightarrow$ `'1983-Q2'`

```
from datetime import datetime

def formula(i1):
  quarter = (i1.month-1)//3 + 1
  return i1.strftime("%Y") +
         "-Q"+f"{quarter:01.0f}"
```

```
YEAR(A1) & "-Q" &
     & ROUNDUP(MONTH(A1)/3, 0)
```

```
Text(i1, "yyyy", "en-US")
& "-Q" &
Text(
  RoundUp(Month(i1) /3, 0),
  "0",
  "en-US")
```

Extract number, convert and round: `'Your Total: $1,2564.45'` $\Longrightarrow$ $12564.5d$

```
from decimal import *

def formula(i1):
  source = Decimal(str(float(
    i1.split("$")[-1].replace(",", ""))))
  delta = Decimal("0.5")
  return float((source / delta)
    .quantize(0, ROUND_CEILING) * delta)
```

```
ROUNDUP(
  NUMBERVALUE(
    RIGHT(A1,
      LEN(A1)-FIND("$", A1))
  ) / 0.5,
  0
) * 0.5
```

```
RoundUp(Value(
  Last(
    Split(i1, "$")
  ).Result,
  "en-US"
) * 2, 0) / 2
```

Fig. 3. Code produced by FlashFill++ for various date-time and rounding scenarios in Python (left), Excel (center), and PowerFx (right) respectively.

due to enumeration blowup, rendering a meet-in-the-middle strategy infeasible. We propose a novel *middle-out synthesis* strategy that uses *cuts* to deal with such non-EI operators.

Consider the number parsing, rounding, and formatting subset of the FlashFill++ DSL below

```
decimal roundNumber := RoundNumber(parseNumber, roundNumDesc)
decimal parseNumber := ParseNumber(substr, locale) | ...
string substr     := ...
```

Fix the input-output example ⟨“`The price is $24.58 and 46 units are available.`” $\mapsto$ 24.00⟩ for the non-terminal `roundNumber`. We first discuss the short-comings of both bottom-up and top-down synthesis in this case.

Top-Down Synthesis using Witness Functions. In FlashMeta-style programming-by-example, the primary deductive tools are *witness functions*. Given a specification in the form of an input-output

(a)

Input	Output
CEDAR AVE & COTTAGE AVE; HORSHAM; 2015-12-11 @ 13:34:52;	Fri, 12PM - 3PM
RT202 PKWY; MONTGOMERY; 2016-01-13 @ 09:05:41-Station:STA18;	Wed, 9AM - 12PM
; UPPER GWYNEDD; 2015-12-11 @ 21:11:18;	Fri, 9PM - 12AM

(b)

```
def derive_value(_input):
    text = _input.split(";")[2]
    part_0 = text.split(" ")[0]; part_1 = text.split(" ")[2][:8]
    date = datetime.datetime.strptime(part_0, "%Y-%m-%d")
    time = datetime.datetime.strptime(part_1, "%H:%M:%S")
    base_value = datetime.timedelta(hours=time.hour, minutes=time.minute,
       seconds=time.second, microseconds=time.microsecond)
    delta_value = datetime.timedelta(hours=3)
    time_str = (time - base_value % delta_value).strftime("%#I%p")
    rounded_up_next = (time - base_value % delta_value) + delta_value
    computed_value = time_str + "-" + rounded_up_next.strftime("%#I%p")
    return date.strftime("%a") + ", " + computed_value
```

Fig. 4. (a) A task to map the 911-call logs in the *Input* column to weekday/time buckets in the *Output* column. (b) Python function synthesized by our approach for this task from just one example.

example $i \mapsto o$ and a top-level operator F, a witness function for position k generates a sub-specification for the kth parameter for F. For example, if the top-level operator is $\texttt{concat}(N_1, N_2)$ and the input-output example is $i \mapsto$ "abc", the witness function for the 1st position will return {"a", "ab"} (assuming we do not consider the trivial case of appending an empty string). In the example we are considering, writing a witness functions for the RoundNumber operator is not as straight-forward—there are an infinite set of numbers that can round to 24.00. A standard top-down procedure cannot handle this infinite-width witness function.

Bottom-Up Synthesis. The other major paradigm for programming-by-example is bottom-up synthesis: here, the synthesizer starts enumerating programs – starting from constants and iteratively applying operators from the grammar on the previously generated programs – and checks if any of the enumerated programs satisfies the given input-output example. An efficient bottom-up synthesizer will avoid enumerating all programs using *observational equivalence*—that is, it only generate programs that produce different outputs for the given input. In our running example, the synthesizer will begin by generating `substr` sub-programs and concrete values for `locale` and `roundNumberDesc`. However, enumerating such sub-programs is expensive—the fragment of the DSL reachable from the non-terminal `substr` is large. In fact, string operations like `substr` are best handled using witness functions.

Middle-Out Synthesis using Cuts. Examining our input-output example by hand, it is easy to see that in any valid program the output of `ParseNumber` should be derived from a numerical substring in the input. Intuitively, it does not matter what or how complex the `substr` sub-program is; we can be confident that the output of the `ParseNumber` will be either 24.58 or 46. Cuts capture this simple intuition—for a given input-output example $i \mapsto o$ and a non-terminal N that expands to $f(N_1, N_2)$, the *cut* for N_1 (say) in the context of N will be a set of values $\{o_1, o_2, \ldots, o_n\}$ such that in any desired program P generated by N, the output of the sub-program corresponding to N_1 will be one of the o_k for $k \in \{1, 2, \ldots, n\}$.

Given that the output of `ParseNumber` will be either 24.58 or 46, the synthesizer has two sub-tasks for the 24.58 case (the 46 case will be similar): (a) synthesizing a program for *parseNumber* for the example $i \mapsto 24.58$, and (b) synthesizing a program for *roundNumber* for the example

$i \mapsto 24.00$ using a modified DSL, which is generated dynamically in middle-out synthesis, where `parseNumber` $\rightarrow 24.58$ is the only rule whose head is `parseNumber`. Both sub-tasks can now be recursively solved, possibly using either the top-down strategy or the bottom-up strategy.

Guarded Context-Free Grammars. The larger program space resulting from an extended DSL with a wide range of operators poses both efficiency and ranking challenges. However, human programmers often encounter the same challenge when writing their own implementations and decide between these operators and programs using simple rules of thumb, which can be leveraged both for improving search efficiency and ranking. For example, "if a task can be done using a date-time function, do not use string transformation functions" or "if a task can be done using string indexing, do not use regular expressions". To mimic this kind of coarse reasoning, we introduce the notion of gDSLs. In a gDSL, the production rules for each non-terminal are ordered (with a partial order $\models$), with production rules earlier in the order preferred to ones later in the order. For example, the rule `concat := segment` $\models$ `Concat(segment, concat)` expresses that we always prefer programs that do not use the `Concat` operation to ones that do. During synthesis for a gDSL rule $N \rightarrow \alpha \models \beta$ the branch β is explored only if the branch α fails to produce a program. This greatly improves the performance of synthesis and the FlashFill++ synthesis times are competitive with other synthesis techniques that work with significantly smaller DSLs.

Apart from improving the efficiency of search, gDSLs also simplify the task of writing ranking functions. Intuitively, the precedence in the guarded rules induce a ranking on programs, and any additional ranking function only needs to order the remaining incomparable programs. Precedences rank programs by a *lexicographic path ordering* (LPO) [Dershowitz and Jouannaud 1990], and our final ranker will be a lexicographic combination of LPO and base arithmetic ranker – such program rankers have not been used in program synthesis before.

3 BACKGROUND: PROGRAMMING BY EXAMPLE

We now define the problem of programming-by-example and discuss common solutions.

3.1 Domain-Specific Languages

We use *domain-specific languages* (DSLs) to specify the set of target programs for a synthesizer. Formally, a DSL D is given by $\langle \mathcal{N}, \mathcal{T}, \mathcal{F}, \mathcal{R}, \mathcal{V}_{\text{in}}, v_{\text{out}} \rangle$ where:

- $\mathcal{N}$ is a finite set of *non-terminal symbols* (or non-terminals). The symbol v_{out} is a special *start non-terminal* in $\mathcal{N}$ that represents the output of a program in the DSL.
- $\mathcal{T}$ is a set of *terminal symbols* (or terminals) that is partitioned as $\mathcal{V}_{\text{in}} \cup \mathcal{O}$ into *inputs* $\mathcal{V}_{\text{in}}$ and *values* $\mathcal{O}$. The set $\mathcal{V}_{\text{in}}$ contains special terminals, $\mathsf{in}_1, \mathsf{in}_2, \ldots$, that represent the input symbols in a program of the DSL. The set $\mathcal{O}$ contains constant values.
- $\mathcal{F}$ is a finite set of *function symbols* (or operations). Each operation $\mathsf{f} \in \mathcal{F}$ has a fixed arity $\mathsf{Arity}(\mathsf{f})$. The semantics of f, denoted by $[\![\mathsf{f}]\!]$, is a mapping from $\mathcal{O}^{\mathsf{Arity}(\mathsf{f})}$ to $\mathcal{O}$.
- $\mathcal{R}$ is a set of *rules* of the form $N \rightarrow \mathsf{f}(v_1, \ldots, v_k)$ or $N \rightarrow v_0$ where $N \in \mathcal{N}$, $\mathsf{f} \in \mathcal{F}$, $v_0 \in \mathcal{T}$, and $v_1, \ldots, v_k \in \mathcal{N} \cup \mathcal{T}$.

The formalism above is *untyped* for ease of reading. In practice, the FlashFill++ DSL is typed— values can be integers, floats, strings, Booleans, and date-time objects, and each non-terminal, terminal, and operator have specific type signatures.

Every terminal or nonterminal v generates a set $\mathcal{L}(v)$ of *programs* defined recursively as follows: (a) $\mathcal{L}(\mathsf{in}_k) = \{\mathsf{in}_k\}$ for all input symbols $\mathsf{in}_k \in \mathcal{V}_{\text{in}}$, (b) $\mathcal{L}(o) = \{o\}$ for all values $o \in \mathcal{O}$, and (c) $\mathcal{L}(N) = \{\mathsf{f}(P_1, \ldots, P_n) \mid N \rightarrow \mathsf{f}(v_1, \ldots, v_n) \in \mathcal{R}, \forall i.P_i \in \mathcal{L}(v_i)\} \cup \{P \mid N \rightarrow v \in \mathcal{R}, v \in \mathcal{T}, P \in \mathcal{L}(v)\}$. The set of programs defined by the whole DSL $\mathcal{L}(\mathsf{D})$ is $\mathcal{L}(v_{\text{out}})$.

Example 3.1. The following is a simple DSL D_A for affine arithmetic expressions over naturals $\mathbb{N}$: (a) Here, the terminals $\mathcal{T}$ are $\{\mathtt{input}_1, \mathtt{input}_2, 0, 1, 2, \ldots\}$, with $\mathtt{input}_1, \mathtt{input}_2$ being the special input terminals in $\mathcal{V}_{\mathsf{in}}$. (b) The non-terminals $\mathcal{N}$ are $\{\mathtt{output}, \mathtt{addend}, \mathtt{const}\}$, with $\mathtt{output}$ being the output symbol v_{out}. (c) The operators $\mathcal{F}$ are $\mathtt{Plus}$ and $\mathtt{Times}$. (d) The rules $\mathcal{R}$ are given by $\{\mathtt{output} \rightarrow \mathtt{Plus}(\mathtt{addend}, \mathtt{output}), \mathtt{output} \rightarrow \mathtt{const}, \mathtt{addend} \rightarrow \mathtt{Times}(\mathtt{const}, \mathtt{input}_1), \mathtt{addend} \rightarrow \mathtt{Times}(\mathtt{const}, \mathtt{input}_2), \mathtt{const} \rightarrow 0 \mid 1 \mid 2 \mid \ldots\}$. Note that the declaration `uint const;` in the listing is shorthand for a set of rules for the form $\mathtt{const} \rightarrow k$ for each $k \in \mathbb{N}$. $\mathtt{Plus}(\mathtt{Times}(5, \mathtt{input}_1), 3)$ is a sample program in $\mathcal{L}(\mathsf{D}_A)$. □

```
@input uint input1, input2;
@start uint output := Plus(addend, output) | const;
       uint addend := Times(const, input1) | Times(const, input2);
       uint const;
```

3.2 Synthesis Tasks and Solutions

A *state* S is a valuation of all input symbols in a DSL, i.e., $S = \{\mathsf{in}_1 \mapsto o_1, \ldots, \mathsf{in}_k \mapsto o_k\}$ where in_i are input symbols and o_i are values. An *example* Ex is a pair $S \mapsto o$ of a state S and value o.

A *synthesis task* $\langle \alpha, \mathcal{R}, S \mapsto o \rangle$ is given by: (a) a term α, which is either a nonterminal, terminal, or right-hand side of a rule, (b) a set $\mathcal{R}$ of rules, and (c) an example $S \mapsto o$. A *solution* of the synthesis task $\langle \alpha, \mathcal{R}, S \mapsto o \rangle$ is a program P such that: (a) $[\![P]\!](S) = o$, and (b) $P \in \mathcal{L}(\alpha)$. Here, $[\![P]\!] : \mathcal{O}^{\mathcal{V}_{\mathsf{in}}} \mapsto \mathcal{O}$ represents the standard semantics of a program. Formally, $[\![P]\!](S)$ is recursively defined as: (1) $[\![\mathsf{in}_i]\!](S) = S(\mathsf{in}_i)$, (2) $[\![o]\!](S) = o$ for every value o, (3) $[\![\mathsf{f}(v_1, v_2)]\!](S) = [\![\mathsf{f}]\!]([\![v_1]\!](S), [\![v_2]\!](S))$ for every operator f. To keep the presentation simple, the synthesis task is defined to contain *one* input-output example. In practice, a synthesis task often involves multiple examples. It is straight-forward to extend our technique for this, as done in our implementation.

Example 3.2. A sample synthesis task for the affine expression DSL D_A is $\langle \mathtt{output}, \mathcal{R}, S \mapsto 7 \rangle$ where $S = \{\mathtt{input}_1 \mapsto 2, \mathtt{input}_2 \mapsto 0\}$. The program $\mathtt{Plus}(\mathtt{Times}(3, \mathtt{input}_1), 1)$ is a solution of this task. Here $\mathtt{Times}$ and $\mathtt{Plus}$ have their usual arithmetic semantics. □

Given a synthesis task $\langle \alpha, \mathcal{R}, S \mapsto o \rangle$, a *synthesizer* generates a program set PS such that every program in the set PS is a solution of the synthesis task, which we denote by the assertion $\mathsf{PS} \models \langle \alpha, \mathcal{R}, S \mapsto o \rangle$. Note that it is vacuously true that $\emptyset \models \langle \alpha, \mathcal{R}, S \mapsto o \rangle$, and so practical synthesizers strive to establish the above assertion for *nonempty* sets PS. The notation $\not\models \langle \alpha, \mathcal{R}, S \mapsto o \rangle$ denotes that there is no nonempty set PS that is a solution for the synthesis task.

3.3 Bottom-Up and Top-Down Synthesis

There are two main approaches for solving the synthesis task: bottom-up and top-down.

The bottom-up (BU) approach enumerates programs generated by different nonterminals of the grammar and collects the values that those programs compute on the input state S. More precisely, for each nonterminal N', the BU approach computes the *bottom-up value set* $\mathsf{bu}_{N'}(S \mapsto o)$ given by $\{[\![P']\!](S) \mid P' \in \mathcal{L}(N')\}$. These sets are computed starting from the leaf (terminals) of the grammar and moving up to the root (start symbol v_{out}). Success is declared if the output value o is found to be in the set $\mathsf{bu}_{v_{\mathsf{out}}}(S \mapsto o)$. Note that the BU procedure is not guided by the output value o.

The top-down (TD) approach starts with the output value o that needs to be generated at start symbol v_{out}, and for every nonterminal N', it computes the set of values that *flow* to o at v_{out}. More formally, we say a value o' at N' flows to value o at N if either (a) $N \rightarrow f(\ldots, N', \ldots)$ is a grammar rule and $[\![f]\!](o_1, \ldots, o', \ldots, o_k) = o$ for some values $o_1, \ldots, o_k$, or (b) there exist o'' and N'' s.t. o' at N' flows to o'' at N'' and o'' at N'' flows to o at N. For each nonterminal N', the

TD approach computes a *top-down value set* $\texttt{td}_{N'}(S \mapsto o)$ that contains the values o' at N' that flow into the value o at v_{out}. The top-down value sets are computed using *witness functions*. An operator f with arity n is associated with n witness functions - one for each argument position. The k-th witness function, $\mathsf{WF}_{\mathsf{f},k} : O^k \mapsto 2^O$, for operator f maps the desired output and $k-1$ values for previous arguments to possible values for the k-th argument. Such a parameterized collection of witness functions is *sound* (*complete*) if $o_k \in \mathsf{WF}_{\mathsf{f},k}(o, o_1, \ldots, o_{k-1})$ for $k = 1, \ldots, n$ implies (is implied by) $[\![\mathsf{f}]\!](o_1, \ldots, o_n) = o$. For example, if the top-level operator is Plus and the output is 7, the possible arguments for Plus would be $(0, 7), (1, 6), (2, 5), \ldots$, and hence, $\mathsf{WF}_{\texttt{Plus},1}(7) = \{0, 1, 2, \ldots\}$ and $\mathsf{WF}_{\texttt{Plus},2}(7, x) = \{7 - x\}$.

Top-down and bottom-up synthesis are both efficient and practical in different scenarios. TD synthesis performs well when operators are *effectively invertible* (EI), i.e., the witness functions $\mathsf{WF}_{\mathsf{f},k}$ for each operator f produces sets that are finite and manageable in size. BU synthesis performs well when the grammar is *effectively enumerable* (EE), i.e., both the number of constants in the grammar is bounded and the number of different intermediate values produced is manageable. Note that the focus in this paper is exclusively on synthesis approaches based on concrete values: the semantics and witness functions are given on concrete values, and not abstract values or types. Abstractions and types can make top-down strategies work on grammars with non-EI operators, for example [Feng et al. 2017; Polikarpova et al. 2016], but require additional machinery, such as type systems, abstract domains, and constraint solvers.

Example 3.3. The affine expression DSL from Example 3.1 is neither effectively invertible nor effectively enumerable. The operator Plus has a witness function $\mathsf{WF}_{\texttt{Plus},1}$ which produces a set of size $n+1$ for an example $S \mapsto n$. Further, it contains an infinite number of constants (i.e., the non-negative integers) which make bottom-up approach infeasible. In the FlashFill++ DSL, many string operators are not effectively invertible. E.g., the LowerCase operator's witness function that can produce a set that is exponential in the input's length: $\mathsf{WF}_{\texttt{LowerCase},1}$(`abc') produces a set of size 8, i.e., {`ABC', `ABc', `AbC', `aBC', `abC', `aBc', `Abc', `abc'}. □

In Section 4, we introduce *cuts* that can be used to decompose the synthesis problem to enable *middle-out synthesis*, which can learn over *deep* DSLs – DSLs that can generate programs with large depth – where neither bottom-up nor top-down is feasible. In Section 5, we introduce gDSLs, which provide a precedence-based mechanism to help learning scale to *broad* DSLs – DSLs with several options for a single nonterminal.

4 CUTS AND MIDDLE-OUT SYNTHESIS

Top-down and bottom-up approaches, as well as their combination, struggle to scale to large DSLs. Cuts can help scale synthesis. If we want to generate a value o at nonterminal N, and another nonterminal N' is on the path from N to the terminals, then a cut at N' returns values to generate at N' that can help with generating o at N.

Definition 4.1 (Cuts). Given a synthesis task $\langle N, \mathcal{R}, S \mapsto o\rangle$ and a non-terminal $N' \in \mathcal{N}$, a *cut* $\mathsf{Cut}_{N',N}$ *for* N' *in the context* N, maps an example, $S \mapsto o$, to a set of values. Such a function is *complete* if for every solution P for the task $\langle N, \mathcal{R}, S \mapsto o\rangle$, whenever P contains a sub-program $P' \in \mathcal{L}(N')$, then $[\![P']\!](S) \in \mathsf{Cut}_{N',N}(S \mapsto o)$.

Note that N need not be the start symbol of the grammar and o need not be the original output value in the input-output example. Typically, we define cuts $\mathsf{Cut}_{N',N}$ when $N \to f(N', N'')$ is a grammar rule. Such a cut can be used in place of the witness function for the first argument of f. Let us illustrate cuts through an example.

$$\text{MO.Cut}\quad \frac{\mathsf{PS}_1 \models \langle N_1, \mathcal{R}, S \mapsto o_1 \rangle \qquad \mathsf{PS}_2 \models \langle N, \mathcal{R} \texttt{ with } N_1 \rightarrow o_1, S \mapsto o \rangle}{\bigcup_{o_1} \mathsf{PS}_2[o_1 \mapsto \mathsf{PS}_1] \models \langle N, \mathcal{R}, S \mapsto o \rangle} \quad \text{if } o_1 \in \mathsf{Cut}_{N_1,N}(S \mapsto o)$$

Fig. 5. The cut inference rule that enables middle-out program synthesis.

Example 4.2. Consider again the synthesis task from Section 2.2, where the input-output example was $S \mapsto 24.00$ and the input state S was $\langle \texttt{in}_1 \mapsto$ "`The price is $24.58 and 46 units are available.`" $\rangle$. Suppose we want to synthesize a program from this example starting from the nonterminal `roundNumber` of the FlashFill++ DSL (Figure 7). One potential cut for `parseNumber` in the context `roundNumber` could work by scanning the input for any maximal substrings that are numerical constants and returning them (as a number). Here, it would return $\{24.58, 46\}$. A more sophisticated cut could additionally look at the output 24.00 and only return the set $\{24.58\}$, as it is the only value in the string that can be rounded down to 24.00. These cuts are not complete as they do not include 24.5, which can also be extracted from the input and rounded to 24. □

Recall that we model synthesizers as generating nonempty program sets PS and asserting $\mathsf{PS} \models \langle \alpha, \mathcal{R}, S \mapsto o \rangle$. Figure 5 presents a new inference rule, called the *cut rule*, that can be used to generate such assertions. This rule can be used in conjunction with any synthesizer (such as those based on top-down or bottom-up approach). The cut rule uses a cut for a nonterminal N_1 in the context of N to decompose the overall synthesis task $\langle N, \mathcal{R}, S \mapsto o \rangle$ into two subtasks $\langle N_1, \mathcal{R}, S \mapsto o_1 \rangle$ and $\langle N, \mathcal{R} \texttt{ with } N_1 \rightarrow o_1, S \mapsto o \rangle$. The first, or *inner*, subtask tries to find a program in $\mathcal{L}(N_1)$ that maps S to o_1, whereas the second, or *outer*, subtask tries to find a program in $\mathcal{L}(N)$ that maps S to o *assuming that we have a program to maps S to* o_1. The notation $\mathcal{R} \texttt{ with } N_1 \rightarrow o_1$ simply means we remove all old rules in $\mathcal{R}$ of the form $N_1 \rightarrow \alpha$ and only have one rule $N_1 \rightarrow o_1$. The cut rule also shows how the solutions to the two subtasks are combined to generate a solution for the original synthesis task.

THEOREM 4.3. *[Soundness] If program sets* PS_1 *and* PS_2 *are such that* $\mathsf{PS}_1 \models \langle N_1, \mathcal{R}, S \mapsto o_1 \rangle$ *and* $\mathsf{PS}_2 \models \langle N, \mathcal{R} \texttt{ with } N_1 \rightarrow o_1, S \mapsto o \rangle$*, then* $\mathsf{PS}_2[o_1 \mapsto \mathsf{PS}_1] \models \langle N, \mathcal{R}, S \mapsto o \rangle$*. Furthermore, [completeness] if a program P is a solution for the synthesis task $\langle N, \mathcal{R}, S \mapsto o \rangle$ and the program P contains a subprogram $P_1 \in \mathcal{L}(N')$ that maps the input S to a value o_1 (i.e., $[\![P_1]\!](S) = o_1$), then o_1 will be in the cut* $\mathsf{Cut}_{N_1,N}(S \mapsto o)$ assuming that the cut is complete*, and moreover, the program $P[P_1 \mapsto o_1]$ is a solution for the task $\langle N, \mathcal{R} \texttt{ with } N_1 \rightarrow o_1, S \mapsto o \rangle$.*

We use the term *middle-out synthesis* to describe the synthesis approach that uses the Rule MO.Cut to perform synthesis. Note that the subproblems created by Rule MO.Cut can be solved using either the top-down approach, or the bottom-up approach, or the middle-out approach, or a hybrid combination of the approaches. One common strategy is: after applying Rule MO.Cut, we solve the outer subtask using bottom-up or hybrid approach, and for each o_1 for which the outer has a solution, we solve the inner subtask using the top-down or hybrid approach. Note that the cut rule can be used multiple times to solve a synthesis task.

Example 4.4. Consider the synthesis task $\langle \texttt{roundNumber}, \mathcal{R}, S \mapsto 24.00 \rangle$ from Example 4.2. Using the fact that 24.58 was in the cut for `parseNumber`, we can use MO.Cut rule from Figure 5 and get the subtasks $\langle \texttt{parseNumber}, \mathcal{R}, S \mapsto 24.58 \rangle$ and $\langle \texttt{roundNumber}, \mathcal{R}', S \mapsto 24.00 \rangle$, where $\mathcal{R}'$ is $\mathcal{R} \texttt{ with } \texttt{parseNumber} \rightarrow 24.58$. The second subproblem now has only *one* rule for `parseNumber`, which directly generates 24.58. The first subproblem now has to generate 24.58 from the input, and the second subproblem has to round 24.58 to 24.00. □

4.1 Generalizing Top-Down and Bottom-Up Synthesis

Examining the top-down and bottom-up synthesis approaches closely, it can be seen that top-down synthesis is purely *output driven* and bottom-up synthesis is purely *input driven.* Cuts neatly generalize both these approaches, allowing us the possibility to use a set of values influenced by both: (a) the set B of values reachable through forward semantics from the input values (bottom-up search), and (b) the set T of values reachable through inverse semantics from the output values (top-down search).

Recall that the set $\mathtt{bu}_{N'}$ is the set of values that bottom-up enumeration generates corresponding to nonterminal N' and the set $\mathtt{td}_{N,N'}$ is the set of values that arise at N' by repeatedly applying (precise) witness functions starting from N. Let *real value set* $\mathtt{rv}_{N,N'}(S \mapsto o)$ be the set of values $[\![P']\!](S)$ where $P' \in \mathcal{L}(N')$ is a sub-program of a program $P \in \mathcal{L}(N)$ such that $[\![P]\!](S) = o$. Clearly, it can be seen that $\mathtt{rv}_{N,N'}(S \mapsto o) \subseteq \mathtt{td}_{N,N'}(S \mapsto o) \cap \mathtt{bu}_{N'}(S \mapsto o)$. Restating the definition of complete cuts, a cut is complete if and only if the set it returns is a superset of $\mathtt{rv}_{N,N'}(S \mapsto o)$. Both top-down and bottom-up search for synthesis are special cases of cut-based middle-out synthesis.

Theorem 4.5. *[Cuts generalize top-down and bottom-up value sets.] Given a synthesis problem* $\langle N, \mathcal{R}, S \mapsto o\rangle$ *and a non-terminal* N'*, both the functions* $\mathtt{bu}_{N'}$ *and* $\mathtt{td}_{N,N'}$ *are complete cuts for the non-terminal* N' *in the context of* N.

In the light of this theorem, restricting the cut in the middle-out synthesis rule from Figure 5 to only being $\mathtt{bu}_{N'}$ produces the state-of-the-art combination of top-down and bottom-up synthesis Duet [Lee 2021]. While the above theorem states that TD and BU analyses produce complete cuts, not all complete cuts are (overapproximations of) top-down value set or bottom-up value set.

Example 4.6. Building on Example 4.2, consider now the example $S \mapsto 24.58$, but we want to synthesize a program from this example starting from the nonterminal `parseNumber`, which has a rule `parseNumber` $\rightarrow$ `ParseNumber(substr, locale)`. One potential cut for `substr` in the context `parseNumber` could be obtained by scanning the input string for any substrings that are numerical constants and returning them (as a string). Here it returns a set containing "24.58" and "46" and all substrings of these two strings that are valid numbers. This complete cut is not an overapproximation of the bottom-up values that `substr` can generate since there are many more substrings in an input. However, it is complete in the context `parseNumber` because these are the only strings that can be parsed as numbers. □

4.2 Computing Cuts

We can use top-down value sets or bottom-up value sets as the cuts, as shown in Theorem 4.5; however, if we do that, we only replicate top-down, bottom-up, and Duet's meet-in-the-middle synthesis [Lee 2021]. DSL designers can provide more refined cuts that would enable going beyond current methods. How can designers author more refined cuts? For most operators, using witness functions to perform top-down synthesis might suffice. Specialized cuts are only required when we have not-effectively-invertible operators that have either no witness function or very inefficient witness functions.

In practice, DSL designers do not necessarily have to use *complete* cuts. Looking back at Example 4.6, the set {"24.58", "46"} is a *reasonable*, but incomplete cut, because any program that extracts a number from a strict substring, say "4.5", of these two strings would be a contrived program. Cuts are reminiscent of *interpolants* or *invariants* from program analysis: a cut at N' in the context of N is the denotation of an invariant that holds true at N' for all programs that compute the desired output at N. We next describe a few cut functions used in FlashFill++ along with the DSL designer's intuition about the DSL that helped construct that cut function.

Cut for LowerCase. When computing a cut for a nonterminal N' in context of nonterminal N, we need to consider all paths from N' to N in the grammar. We focus on one path at a time, and take a union if there are multiple paths. Consider the grammar rule `single` := `LowerCase(concat)` in the FlashFill++ DSL (Figure 7), and ignore the other paths between nonterminals `single` and `concat` for now. Clearly, the witness function is given as:

$$\mathsf{WF}_{\mathsf{LowerCase},1}(y) = \{x \mid \mathtt{lowercase}(x) = y\}$$

The returned set contains $2^{\mathtt{len}(y)}$ strings. In contrast, the cut could be much smaller. Ignoring other paths between the two nonterminals, one possible cut is:

$$\mathsf{Cut}_{\mathsf{concat},\mathsf{single}}(S \mapsto y) = \{x \mid \mathtt{lowercase}(x) = y \text{ and each char of } x \text{ is in some input in } S\}.$$

Here the DSL designer uses their knowledge that *the non-terminal* `concat` *can only generate strings whose characters are in some input.* One exception to this invariance rule are strings that are generated as "constant strings", and hence this cut is not complete, but it may be a reasonable compromise between completeness and efficiency. An alternative cut could be:

$$\{x \mid \mathtt{lowercase}(x) = y \text{ and for each char } x[i] \text{ of } x, x[i] \text{ is in some input or } x[i] == y[i]\}.$$

Finally, the designer can further refine the cut to only include those strings that contain large chunks of substrings of some input. Consider input `'Amal Ahmed <AMAL@CCS.NEU.EDU>'` and output `'amal'`. The witness function would return 16 values, all variations of the string `'amal'` with each letter optionally capitalized. However, a cut would only return 2 values `'Amal'` and `'AMAL'`, i.e., those variations that occur contiguously in the input. The same kind of reasoning can be used on all other paths from `concat` to `single` that go via the operators `UpperCase` and `ProperCase`, and thus we can get a cut for `concat` in the context of `single`.

Cut for Concat. Consider the grammar rule `concat` := `segment|Concat(segment, concat)` in the FlashFill++ DSL (Figure 7). The witness function for (the first argument of) `Concat` is given by

$$\mathsf{WF}_{\mathsf{Concat},1}(y) = \{x \mid x \text{ is a prefix of } y\}$$

The DSL designer knows the invariant that *every string generated by* `segment` *is either a substring of an input, or a string representation of date or number, or a constant string.* So, a possible cut, $\mathsf{Cut}_{\mathsf{segment},\mathsf{concat}}(S \mapsto y)$, could be:

$$\{x \mid x \text{ is maximal prefix of } y \text{ either contained in some input or a number or a date}\}.$$

This cut is not complete since `segment` could generate a constant string, but the DSL designer may prefer to use this cut and fallback on the witness function only if the above choices fail to work.

Cut for RoundNumber and FormatDateTime. Example 4.2 presented the cut for `parseNumber` in the context `roundNumber`. The witness function for `roundNumber` on the input $S \mapsto 24.00$ will have to return an infinite set of floating point values that can all round to 24.00. However, the DSL designer knows that `parseNumber` *can only generate a number that occurs in the input, i.e.,* 24.58 *or* 46*, and furthermore, numbers such as* 4.5 *are not reasonable choices.* Hence, the designer can pick the cut $\mathsf{Cut}_{\mathsf{parseNumber},\mathsf{roundNumber}}(S \mapsto y)$:

$$\{x \mid x \text{ is a maximally long number extracted from a substring of an input}\}.$$

Here the DSL designer used their knowledge about the form of values that a nonterminal can generate to construct a cut. Another such example is the cut, $\mathsf{Cut}_{\mathsf{asDate},\mathsf{formatDate}}(S \mapsto y)$, which can be computed as:

$$\{x \mid x \text{ is a maximally long datetime value extracted from a substring of an input}\}.$$

Cut for Const in the Affine Grammar. The DSL designer of arithmetic expressions given in Example 3.1 can provide a cut for `const` in the context `output` by noting *the monotonicity invariant: values computed by subexpressions are smaller than values computed by the whole expression.* Consider the input state $S = \langle \mathsf{in}_1 \mapsto 2, \mathsf{in}_2 \mapsto 0\rangle$ and the input-output example $S \mapsto 7$. Since the output is 7, we can restrict the potential values for the coefficient of in_1 (which is 2) to at most 3 values, i.e., $\{0, 1, 2, 3\}$ as any larger value will make the product exceed the value 7. Thus, the cut, $\mathrm{Cut}_{\mathsf{const},\mathsf{output}}(S \mapsto y)$, is computed as:

$$\{x \in \mathbb{N} \mid 0 \leq x \leq \lfloor \frac{y}{S[\mathsf{in}_1]} \rfloor\}.$$

Using this cut we can now perform bottom-up synthesis, whereas it wasn't possible before since there are an infinite set of possible values for the non-terminal `const` and the grammar is not effectively enumerable.

Whenever there is a nontrivial invariant that holds for all values that can be generated at a certain nonterminal, we can usually exploit that invariant to design a cut for that nonterminal. While we have focused mainly on the FlashFill++ DSL to illustrate this process of designing good cuts, the ideas extend to DSLs for any other domain.

5 PRECEDENCE IN DOMAIN-SPECIFIC LANGUAGES

We first define the problem of synthesis in presence of precedence over operators. We then introduce gDSLs, which extend the notion of DSL (Section 3.1) with precedence. We then present the gDSL for FlashFill++ and inference rules that solve the PBE synthesis problem over gDSLs.

5.1 Synthesis with Preference

DSL designers who want to translate programs generated in a DSL into a popular target language, such as Python, typically want the DSL to contain all operators from the target language libraries. These operators are often redundant. For example, substrings of a string can be extracted using absolute index positions, or regular expressions, or finding locations of constant substrings in the string, or splitting a string by certain delimiters. In such cases, whenever a task is achievable in many different ways, we get a so-called *broad* DSL. For broad DSLs, DSL designers often have a preference for which operators to use. For example, they may prefer `split` over `find`, which they may prefer in turn over using regular expressions or absolute indices. As another example, DSL designers may prefer transforming a string containing "Jan" to "January" by treating the substring around "Jan" in the input as a datetime object and working with them, rather than generating "January" by concatenating "Jan" with a constant string "uary".

Example 5.1. Consider the task of extracting the second column from a line in a comma-separated values (CSV) file, specified by the input-output example `'WA, Olympia, UTC-8'` $\mapsto$ `'Olympia'`. In the FlashFill++ DSL, this can be done using the program `Split(x, ',', 2)`, where x is the input, or using the program `Slice(x, Find(x, ',', 1, 0), Find(x, ',', 2, 0))`. A traditional VSA-based synthesizer would (possibly implicitly) produce both programs, assign scores to both using a ranking function, and return the better ranked one. However, typically we strictly prefer programs that use the `Split` operator over the `Slice` operator. Ideally, a synthesizer should not even examine `Slice` programs when an equivalent `Split` program exists. Similar preference is also seen in Python programmers who often prefer to use `str.split` over `regex.find` or `str.find`. □

This motivates the need to perform synthesis over a broad DSL where there is preference over operators. Suppose a DSL designer has a DSL and a preference over operators and terminals. Let $\Sigma := \mathcal{F} \cup \mathcal{T}$ be the collection of all operators and terminal symbols in the DSL and let $>_\Sigma$ be a precedence relation on the symbols $\mathsf{f} \in \Sigma$. We make the assumption that (A1) the DSL designer

only provides precedence over operators that occur as alternates for the same nonterminal; that is, if the designer sets $\mathsf{f}_1 \succ_\Sigma \mathsf{f}_2$, then $N \to \mathsf{f}_1(\dots) \in \mathcal{R}$ and $N \to \mathsf{f}_2(\dots) \in \mathcal{R}$ are two rules with the same nonterminal N in the grammar. We also assume that (A2) the relation $\succ_\Sigma$ is a strict partial order (irreflexive, asymmetric, transitive) to ensure that the DSL designers preference is consistent.

We first need to formalize what it means for a synthesizer to satisfy the operator precedence $\succ_\Sigma$ provided by the DSL designer. For this, we need to lift $\succ_\Sigma$ to a precedence on the set of programs P generated by the DSL. However, this is not easy since it is not clear which of $\mathsf{f}_1(\mathsf{f}_2(\mathtt{in}))$ or $\mathsf{f}'_1(\mathsf{f}'_2(\mathtt{in}))$ to prefer if the DSL designer says $\mathsf{f}_1 \succ_\Sigma \mathsf{f}'_1$ and $\mathsf{f}'_2 \succ_\Sigma \mathsf{f}_2$. We resolve this issue by saying that the preference for the operator occurring "above" in the program is more important than anything below. In the above example, we give more weight to $\mathsf{f}_1 \succ_\Sigma \mathsf{f}'_1$ and hence we want $\mathsf{f}_1(\mathsf{f}_2(\mathtt{in}))$ to be preferred over $\mathsf{f}'_1(\mathsf{f}'_2(\mathtt{in}))$. This follows the intuition that operators at the top in, say, the FlashFill++ DSL, such as `Concat`, `FormatNumber`, or `FormatDateTime`, are more influential in determining the high-level strategy for solving a task than operators at the bottom, such as `Split` or `Slice`. Hence, we extend the user provided $\succ_\Sigma$ to $\succ^e_\Sigma \supseteq \succ_\Sigma$ so that whenever $N_0 \to \mathsf{f}_1(\dots, N_1, \dots)$ and $N_1 \to \mathsf{f}_2(\dots)$ are rules in the DSL, then $\mathsf{f}_1 \succ^e_\Sigma \mathsf{f}_2$.

Example 5.2. Consider the synthesis task from Example 4.2 of generating `'24.00'` from the input string. One possible program is `Concat`(p_1, `'.00'`), where p_1 is a subprogram that extracts `'24'` from the input string. A second possible program is `Segment(FormatNumber(`p_2, fmt_desc) that generates `'24.00'` by formatting a number computed by program p_2. Here, `Segment` is a dummy identity operator having higher preference than `Concat` in the FlashFill++ gDSL (Figure 7). In the FlashFill++ DSL, the second program is preferred since it does not use concatenation, irrespective of how p_1 and p_2 work—at the top-level concatenation is strictly less preferred. Note that here p_1 is likely to be significantly smaller and simpler than p_2 as it is just extracting the string `'24'`, while p_2 is extracting a number and then rounding it. A traditional arithmetic ranking function (as used in FlashFill and FlashMeta) intuitively computes the score of programs as a weighted sum of the score of its sub-programs, and hence, will need to be tuned carefully to ensure that the smaller concat program is scored worse than the larger segment program. □

We want the relation $\succ^e_\Sigma$ to be a strict partial order. However, in general, it may not be a strict partial order due to cycles in the grammar (where some N_0 generates a term containing N_0), which again makes $\succ^e_\Sigma$ violate irreflexivity or transitivity. We make the reasonable assumption that there are no cycles since we often limit the depth of terms being synthesized and then the assumption can be satisfied by renaming the nonterminals. Thus, without loss of much generality, we can assume that the extended precedence $\succ^e_\Sigma$ on Σ is a strict partial order.

Now we formalize synthesizing in presence of precedence $\succ_\Sigma$ by lifting the precedence $\succ_\Sigma$ on Σ to $\succ^e_\Sigma$, and then to a preference on programs (trees) over Σ using the well-known *lexicographic path ordering* (LPO), $\succ_{lpo}$, which is defined as follows [Dershowitz and Jouannaud 1990]: Given programs $P_1 := \mathsf{f}_1(P_{11}, \dots, P_{1k})$ and $P_2 := \mathsf{f}_2(P_{21}, \dots, P_{2l})$, we have $P_1 \succ_{lpo} P_2$ if either (a) $\mathsf{f}_1 \succ^e_\Sigma \mathsf{f}_2$ and $P_1 \succ_{lpo} P_{2i}$ for all i, or (b) $\mathsf{f}_1 = \mathsf{f}_2$ and there exists a m s.t. $P_{1i} = P_{2i}$ for $i < m$ and $P_{1m} \succ_{lpo} P_{2m}$, or (c) $P_{1i} \succ_{lpo} P_2$ for some i.

Definition 5.3. Let $\geq_{base}$ be the base ordering to rank programs that are unordered by $\succ_{lpo}$. Given a DSL D with precedence $\succ_\Sigma$ on the set $\Sigma := \mathcal{F} \cup \mathcal{T}$ of all operators and terminal symbols in D, and $\geq_{base}$, the *PBE synthesis with precedence* problem is to find the maximally ranked program by *the lexicographic combination* of $\succ_{lpo}$ and $\geq_{base}$ that satisfies a given example, where $\succ_{lpo}$ is the LPO induced by the extended precedence $\succ^e_\Sigma$.[2]

[2]Given orderings $\succ_1$ and $\succ_2$, the lexicographic combination $\succ_{\succ_1,\succ_2}$ is defined as follows: $s \succ_{\succ_1,\succ_2} t$ if either (a) $s \succ_1 t$, or (b) $s \not\succ_1 t$ and $t \not\succ_1 s$ and $s \succ_2 t$.

We solve the PBE synthesis with precedence problem by extending DSLs to gDSLs and modifying the inference rules to correctly handle the precedence.

5.2 Guarded Domain-Specific Languages

A *guarded domain-specific language (gDSL)* is a DSL $\mathsf{D} = \langle \mathcal{N}, \mathcal{T}, \mathcal{F}, \mathcal{R}, \mathcal{V}_{\text{in}}, v_{\text{out}} \rangle$ where the set $\mathcal{R}$ of rules can additionally contain *guarded rules* of the form $N \to \alpha_1 \models \alpha_2 \models \cdots \models \alpha_k$ where each α_i is either $\mathsf{f}(v_1, \ldots, v_n)$ or v_0, where $\mathsf{f} \in \mathcal{F}$, $v_0 \in \mathcal{T}$, and $v_1, \ldots, v_n \in \mathcal{N} \cup \mathcal{T}$. A non-terminal N can have any number of guarded rules associated with it, each with possibly different values of $k \geq 1$.

The rules in a regular DSL can be viewed as a special case of guarded rules where $k = 1$ (in the definition of guarded production rules above.) When $k > 1$, a guarded rule $N \to \alpha_1 \models \cdots \models \alpha_k$ associated with the nonterminal N has k alternates on the right-hand side that are *ordered*. The i-th alternate α_i yields a (regular) rule $N \to \alpha_i$. We call $N \to \alpha_i$ the *i-th constituent rule* of the original guarded rule. Define $\mathcal{R}^c$ as the collection of all constituent rules of rules in $\mathcal{R}$; that is, $\mathcal{R}^c := \{N \to \alpha_i \mid N \to (\ldots \models \alpha_i \models \ldots) \in \mathcal{R}\}$. We call the (regular) DSL $\mathsf{D}^c := \langle \mathcal{N}, \mathcal{T}, \mathcal{F}, \mathcal{R}^c, \mathcal{V}_{\text{in}}, v_{\text{out}} \rangle$ obtained from a gDSL $\mathsf{D} := \langle \mathcal{N}, \mathcal{T}, \mathcal{F}, \mathcal{R}, \mathcal{V}_{\text{in}}, v_{\text{out}} \rangle$ a *constituent* DSL of the gDSL D.

Given an instance of the PBE synthesis with precedence problem, we can annotate the given DSL with precedence to get a gDSL. We assume that the precedence relation $>_\Sigma$ satisfies Assumptions (A1) and (A2). Furthermore, we also assume that (A3) the precedence is a *series-parallel partial order* (SPPO) [Béchet et al. 1997]. Under Assumption (A3), the precedence can be encoded in gDSL by introducing new nonterminals where necessary. Specifically, if we have a maximal chain $\mathsf{f}_1 >_\Sigma \mathsf{f}_2 >_\Sigma \ldots >_\Sigma \mathsf{f}_k$ over alternate operators $N \to \mathsf{f}_1(\ldots) | \cdots | \mathsf{f}_k(\ldots)$ in the DSL, then we add a guarded rule $N \to \mathsf{f}_1(\ldots) \models \cdots \models \mathsf{f}_k(\ldots)$. However, if certain alternate operators are left incomparable by the DSL designers, then we introduce a new nonterminal. For example, if $\mathsf{f}_1 >_\Sigma \mathsf{f}_3$ and $\mathsf{f}_2 >_\Sigma \mathsf{f}_3$, but there is no preference between f_1 and f_2, then we introduce a new nonterminal N' and have $N \to N' \models \mathsf{f}_3(\ldots)$ and $N' \to \mathsf{f}_1(\ldots) | \mathsf{f}_2(\ldots)$ in the gDSL. Any SPPO $>_\Sigma$ can thus be encoded in the gDSL.

Example 5.4. Consider the DSL for affine arithmetic from Example 3.1. Suppose the DSL designer wants the precedence $\mathtt{input}_1 >_\Sigma \mathtt{input}_2$. Then, we can replace the two rules for nonterminal addend by a single guarded rule: $\mathtt{addend} \to \mathtt{Times}(\mathtt{const}, \mathtt{input}_1) \models \mathtt{Times}(\mathtt{const}, \mathtt{input}_2)$ to get a gDSL, D^g_A. In the LPO induced by the extension $>^e_\Sigma$ of this preference, the program $P_1 := \mathtt{Plus}(\mathtt{Times}(3, \mathtt{input}_1), 1)$ is preferred over $P_2 := \mathtt{Plus}(\mathtt{Times}(3, \mathtt{input}_2), 7)$. □

Let $\langle v_{\text{out}}, \mathcal{R}, S \mapsto o \rangle$ be a synthesis task, where $\mathcal{R}$ is the rules of a gDSL D. We say a program P is a *solution* for this task if
(a) P is a solution for the task $\langle v_{\text{out}}, \mathcal{R}^c, S \mapsto o \rangle$ (in the constituent DSL D^c), and
(b) for any other P' that is a solution in the constituent DSL, $P' \not>_{lpo} P$, where $>_{lpo}$ is the LPO induced by the extended precedence $>^e_\Sigma$ coming from the guarded rules.

Condition (a) says that P should be a solution for the synthesis problem ignoring the precedence. Condition (b) says that the ordering in the rules should be interpreted as an ordering on programs using the induced LPO, and we should ignore programs smaller in this ordering.

Example 5.5. Consider the gDSL D^g_A and programs P_1, P_2 from Example 5.4. Consider the synthesis task $\langle v_{\text{out}}, \mathcal{R}, \langle \mathtt{input}_1 \mapsto 2, \mathtt{input}_2 \mapsto 0 \rangle \mapsto 7 \rangle$ from Example 3.2, but with $\mathcal{R}$ now coming from the gDSL D^g_A. Both programs, P_1 and P_2, map the input state to 7. However, P_2 is now *not* a solution in D^g_A because there exists P_1 that is preferred. The program $P_3 := 7$ also maps the input state to 7. The (derivations of) P_1 and P_3 are incomparable; and in fact, both are solutions in D^g_A. □

If a solution for the unguarded DSL exists, then there will be a solution that is maximal and hence a solution for the gDSL will exist.

$$\textsc{Guarded.If}\ \frac{\mathsf{PS}_1 \models \langle \alpha_1, \mathcal{R}, S \mapsto o\rangle \quad N \to \alpha_1 \models \alpha_2 \in \mathcal{R}}{\mathsf{PS}_1 \models \langle N, \mathcal{R}, S \mapsto o\rangle} \qquad \textsc{Guarded.Else}\ \frac{\not\models \langle \alpha_1, \mathcal{R}, S \mapsto o\rangle \quad N \to \alpha_1 \models \alpha_2 \in \mathcal{R} \quad \mathsf{PS}_2 \models \langle \alpha_2, \mathcal{R}, S \mapsto o\rangle}{\mathsf{PS}_2 \models \langle N, \mathcal{R}, S \mapsto o\rangle}$$

Fig. 6. Extending top-down synthesis for guarded rules.

Theorem 5.6 (Precedence preserves solvability.). *Let* D *be a gDSL based on precedence* $>_\Sigma$ *that is a strict partial order. Let* $\langle v_{\text{out}}, \mathcal{R}, S \mapsto v\rangle$ *be a synthesis task. This task has a solution in* D *iff there is a solution in* D^c.

If we can compute all solutions for a synthesis task over a gDSL, we can order them by any $\geq_{base}$ ordering to solve the PBE synthesis with precedence problem.

5.3 PBE Synthesis Rules for Guarded DSLs

Figure 6 contains two inference rules that describe how guarded rules are handled in top-down, bottom-up, or middle-out synthesis. If $N \to \alpha_1 \models \alpha_2$ is a guarded rule in $\mathcal{R}$ and we can (recursively) prove $PS_1 \models \langle \alpha_1, \mathcal{R}, S \mapsto o\rangle$, then Rule Guarded.If can be used to assert $PS_1 \models \langle N, \mathcal{R}, S \mapsto o\rangle$. (This assertion will be nontrivial only if $PS_1 \neq \emptyset$.) In case there is no program that solves $\langle \alpha_1, \mathcal{R}, S \mapsto o\rangle$, and we can (recursively) prove $PS_2 \models \langle \alpha_2, \mathcal{R}, S \mapsto o\rangle$, then Rule Guarded.Else can be used to assert $PS_2 \models \langle N, \mathcal{R}, S \mapsto o\rangle$.

Recall that the notation $PS \models \langle N, \mathcal{R}, S \mapsto o\rangle$ simply asserts that every program in PS solves the given synthesis task, and does not require PS to contain *all* solutions. The Rule Guarded.Else has a condition that a certain synthesis task be unsolvable. If we have the ability to compute all solutions (such as, using version-space algebras, or VSAs), then that can help in determining when a problem is unsolvable, but other synthesis approaches that can establish infeasibility can also be used.

Example 5.7. Continuing from Example 5.5, consider the task of generating 7 from inputs 2 and 0. Say we reduce the original task to the proof obligation $X \models \langle \texttt{addend}, \mathcal{R}, S \mapsto x\rangle$, where we pick say 6 for x. Now, we have a guarded rule for `addend` and we can use Rule Guarded.If to get the proof obligation $X \models \langle \texttt{Times(const, input1)}, \mathcal{R}, S \mapsto 6\rangle$. This subtask has a solution $X = \{\texttt{Times(3, input1)}\}$, and hence we will not consider the option `Times(const, input2)`. □

Performing synthesis over the gDSL is equivalent to solving PBE synthesis with precedence.

Theorem 5.8. *Let* $>_\Sigma$ *be a precedence on* $\Sigma := \mathcal{F} \cup \mathcal{T}$ *that satisfies Assumptions (A1), (A2), and (A3). Let* $\mathsf{D} := \langle \mathcal{N}, \mathcal{T}, \mathcal{F}, \mathcal{R}, \mathcal{V}_{\text{in}}, v_{\text{out}}\rangle$ *be a gDSL that encodes* $>_\Sigma$. *Let* $\mathsf{D}^c := \langle \mathcal{N}, \mathcal{T}, \mathcal{F}, \mathcal{R}^c, \mathcal{V}_{\text{in}}, v_{\text{out}}\rangle$ *be its (unguarded) constituent DSL. Let* $\geq_{base}$ *be an ordering on programs and let* $\langle v_{\text{out}}, \mathcal{R}, S \mapsto o\rangle$ *be a synthesis task. Then, the following are equivalent:*
(1) $\{P\} \models \langle v_{\text{out}}, \mathcal{R}, S \mapsto o\rangle$ *and* P *is maximal w.r.t* $\geq_{base}$ *among all such solutions.*
(2) The program P *is a solution in* D^c *for the task* $\langle v_{\text{out}}, \mathcal{R}^c, S \mapsto o\rangle$ *that is maximal w.r.t a lexicographic combination of* $>_{lpo}$ *(induced by* $>_\Sigma$*) and* $\geq_{base}$.

Theorem 5.8 shows that using a ranker $\geq_{base}$ with a gDSL D has the same effect as using a complex ranker (lexicographic combination of $>_{lpo}$ and $\geq_{base}$) with a regular DSL D^c. This shows that our gDSL-based approach solves the PBE synthesis with precedence problem (under some assumptions). Theorem 5.8 also explains why the ranker $\geq_{base}$ used with a gDSL D can be very simple compared to what is needed with a regular DSL D^c. Designing a good complex ranking function has traditionally been very challenging in PBE, taking many developer-months to converge on a good ranking function [Kalyan et al. 2018; Natarajan et al. 2019; Singh and Gulwani 2015]. In contrast, FlashFill++ uses the power of gDSLs (Theorem 5.8) to reduce the requirements on its base ranker, which was developed significantly faster.

```
language FlashFillPP;
@start string output := single |> If(cond, single, output)
// conditions
bool cond := pred |> And(pred, cond);
bool pred := StartsWith(x, matchPattern) |> EndsWith(x, matchPattern)
          |> Contains(x, matchPattern) |> ...;
// single branch
string single := concat | LowerCase(concat) | UpperCase(concat) | ProperCase(concat);
// optional concatenation of substrings
string concat := segment |> Concat(segment, concat)
// substring logic
string segment := substr | formatNumber | formatDate | constStr;
// format substring as a number
string formatNumber := FormatNumber(roundNumber, numFmtDesc);
decimal roundNumber := parseNumber |> RoundNumber(parseNumber, roundNumDesc);
decimal parseNumber := AsNumber(row, columnName) |> ParseNumber(substr, locale);
// format substring as a date
string formatDate := FormatDateTime(asDate, dateFmtDesc);
DateTime asDate := AsDateTime(row, columnName) |> ParseDateTime(substr, parseDateFmtDesc);
// find a substring within the input x
string substr := Split(x, splitDelimiter, splitInstance)
              |> Slice(x, pos, pos)
              |> MatchFull(x, matchPattern, matchInstance);
// find a position within the input x
int pos := End(x) |> Abs(x, absPos)
        |> Find(x, findDelimiter, findInstance, findOffset)
        |> Match(x, matchPattern, matchInstance)
        |> MatchEnd(x, matchPattern, matchInstance);
// literal terminals
FmtNumDescriptor numFmtDesc; RndNumDescriptor roundNumDesc;
FmtDateTimeDescriptor dateFmtDes, parseDateFmtDesc;
string constStr, splitDelimiter, findDelimiter;
int splitInstance, findInstance, matchInstance, findOffset;
Regex matchPattern; int absPos;
```

Fig. 7. A fragment of the gDSL for FlashFill++. | choices are unguarded, |> choices are guarded.

5.4 Guarded DSL for FlashFill++

We now describe the FlashFill++ gDSL and compare it to FlashFill. Figure 7 shows a major part of the DSL. FlashFill++ shares the top level rules that perform conditional statements, case conversion, and string concatenation with FlashFill. Conditionals enable if-then-else logic. The condition is one or more conjunctive predicates based on properties of the input string. Case conversion transforms a string into lower-, upper-, or proper-case. Concatenation concatenates two strings.

Although FlashFill can perform some datetime and number operations using text manipulation (such as "01/01/2020" → "2020" or "10.01" → "10"), it is unable to express other sophisticated datetime and number operations as it does not incorporate operations over those datatypes, but rather treats them as standard strings. For instance, FlashFill cannot get the day of week from a date (such as "01/01/2020" → "Wednesday"), or round up a number (e.g., "10.49" → "10.5"). This motivates us to add new rules to support richer datetime (rules `parseDate` and `formatDate`) and number (rules `parseNumber` and `formatNumber`) transformations.

The next major differences are in the `substr` and `pos` rules. FlashFill has a single `Slice` operator which selects a substring defined by its start and end positions. These positions can be defined either as absolute positions or with the complex `RegPos` operator which finds the kth place in the

string where the left- and right substrings match the two given regular expressions. While this is expressive enough to cover any desired substring selection and all of the operators in FlashFill++ can technically be expressed in terms of it, this introduces a challenge for an industrial synthesizer that targets different languages for code generation: not all platforms of interest support regular expressions natively (e.g. Microsoft Excel's formula language does not support regular expressions). In contrast, when designing FlashFill++ we chose a wider collection of more natural operators that are closer to what developers use in practice when working with target languages – this removes the mismatch between synthesis DSL and code generation target language.

In particular, instead of only allowing substrings to be defined as a `Slice` with their start and end positions, FlashFill++ adds a `Split` operator to select the kth element in a sequence generated by splitting the input string by some delimiters. We also add a `MatchFull` operator to find the kth match of a regular expression. Additionally, in FlashFill++ the `pos` rule replaces the operator `RegPos` (which relies on a pair of regular expressions to identify a position) with a `Find` of a constant string in the input and a `Match`/`MatchEnd` of a regular expression.

Example Guarded Rules in the FlashFill++ DSL. We go over a few cases of guarded rules in FlashFill++ to show they capture natural intuitions and rules of thumb in the string transformation domain.

- *Single segments over concats.* The guarded rule `segment` ⊧ `Concat(segment, concat)` ensures that we try to synthesize programs that generate the whole output at once, before generating programs that produce a prefix and a suffix of the output and then combine them. Whenever such a program exists, this guarded rule potentially saves the exploration of a huge portion of the program space. This single guarded rule plays a crucial role in keeping FlashFill++ performant since witness function of the concat operator produces $2(n-1)$ subtasks, one each for the prefix and suffix at each point where the output string can be split.
- *Splits over slices.* As illustrated Example 5.1, FlashFill++ strictly prefers programs that use the `Split` operator over programs that use the `Slice` operator. Program in data wrangling scenarios very commonly begin by extracting the appropriate part of the input from a delimited file record (CSVs, TSVs, etc). In all such cases, a split program more closely follows the human intuition of "extract the nth column delimited by" as compared to a slice program. Note that the split operator is a "higher level" construct preferred over the "low-level" slice.
- *Input numbers over rounded numbers.* The `RoundNumber` operator in Figure 7 is guarded by `parseNumber`, meaning that we can only generate a program that rounds a number if no number in the input can be used directly to produce the same output.

6 EXPERIMENTAL EVALUATION

We now present our experimental results, including an ablation study and survey-based user study.

6.1 Methodology

We used 3 publicly available benchmark sets – Duet string benchmarks [Lee 2021], Playgol [Cropper 2019], and Prose [PROSE 2022] – covering a range of string transformations, including datetime and number formatting.

We compare FlashFill++ with three systems: two publicly available state-of-the-art synthesis systems that are deployed in productions, namely **FlashFill** [Gulwani 2011] and **SmartFill** [Chen et al. 2021a; Google 2021], and one, **Duet**, from a recent publication [Lee 2021]. To experiment with FlashFill, we implemented it on top of the FlashMeta framework [Polozov and Gulwani 2015]. We carry out our FlashFill, FlashFill++, and Duet experiments on a machine with 2 CPUs & 8GB RAM. To experiment with SmartFill, we employ Google Sheets in Chrome and use it to solve the subset of tasks that are suitable for its spreadsheet environment.

Table 1. Comparing different tools (rows) on the 3 public benchmarks (columns) based on (1) number of benchmarks correctly solved (Columns 2–5) and (2) average number of examples required on the successful benchmarks (Columns 6–8). The total number of benchmarks attempted are shown in brackets.

	Number benchmarks solved				Average #examples required		
	Duet (205)	Playgol (327)	Prose (354)	Total (886)	Duet	Playgol	Prose
FlashFill	139	264	172	575	**1.41**	**1.26**	1.54
FlashFill++	**159**	**307**	**353**	**819**	1.69	1.46	**1.52**
Duet	102	211	166	479	1.97	2.11	2.01
SmartFill	37 (158)	34 (327)	18 (296)	89 (781)	2.75	2.85	2.83

Since SmartFill is not exposed in the Google Sheets API, we rely on browser-automation using Selenium [Selenium 2022] for SmartFill evaluation. Moreover, we remove problematic benchmark tasks and use 158, 327, and 296 tasks from the Duet, Playgol, and Prose benchmarks, respectively, for SmartFill evaluation. The tasks removed were unsuitable for browser automation (e.g. too many rows or new line characters).

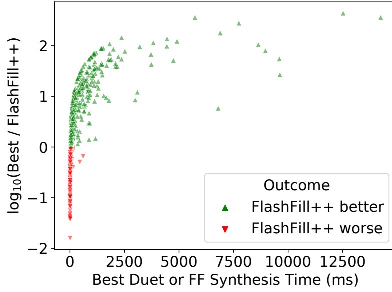
The Duet tool [Lee 2021] accepts a DSL as part of its input. Different Duet benchmarks used slightly different DSLs. For fair comparison, we set a single fixed DSL. We obtained the fixed DSL by taking all commonly-used rules in the string transformation benchmarks of Duet. Second, Duet has some hyperparameters, for which we picked the best setting after some experimentation.

6.2 Expressiveness and Performance

A key feature of FlashFill++ is improved expressiveness. Table 1 (Columns 2–5) shows the number of benchmark tasks that the various tools (rows) can correctly solve. FlashFill++ produces a correct program for most number of benchmarks. FlashFill is limited due to a lack of datetime and number formatting. The DSL used in the Duet tool has no datetime support, and only limited support for number and string formatting. The SmartFill tool is a neural-based general purpose tool and has the weakest numbers here. Since our SmartFill experiments rely on browser-based interaction, it is possible that the underlying synthesizer can solve more benchmarks but these are not exposed to the UI. However, our setup reflects the experience that a user would face.

> Our DSL is expressive, covering a large class of string, datetime and number transformations; thus showing the added value from using cuts and guarded rules.

While increasing expressiveness enables users to accomplish more tasks, there is a risk of reducing performance on existing tasks. To this end, we consider the minimum number of examples required for a synthesizer to learn a correct program. To find that number, we use a counter-example guided (CEGIS) loop which provides the next failing I/O example in every iteration to the synthesizer. We use a time out of 20 seconds. Table 1 Columns 6–8 present the average number of examples the various tools used across the benchmarks where they were successful. Here FlashFill has the best numbers, which indicates that when it works (for string benchmarks), it learns with very few examples. Our tool FlashFill++ takes only slightly more examples on average, partly because datetime and number formatting typically requires more examples for intent disambiguation. For example, the string `'2/3/2020'` can either be the 2nd of March or the 3nd of February. The Duet and SmartFill tools take more examples in general. We emphasize that the performance of FlashFill++ is good here because it solves more problems (presumably much harder instances) and yet it doesn't use too many more examples (the harder instances did not make the averages much worse).



(a) $\log_{10}(\frac{\min(\text{FlashFill,Duet})}{\text{FlashFill++}})$ vs min(FlashFill, Duet).

	Benchmark classes			
	Duet	Playgol	Prose	Overall
FlashFill	689.7	1757.0	734.1	1116.4
FlashFill++	**203.0**	**217.1**	**246.4**	**228.5**
Duet	264.0	558.4	1351.7	766.74

(b) Average time (in ms) taken by the tools (rows) over successful benchmarks from the three sources (columns), and the average for each tool over the entire suite (last column).

Fig. 8. FlashFill++ is generally faster – up to two orders of magnitude – than the best of FlashFill and Duet, and the slowdowns are mostly on fast benchmarks.

> Despite solving more tasks, FlashFill++ continues to require a reasonable number of examples compared to baselines, showing that it generalizes to unseen examples and does not overfit.

We next compare synthesis times (averaged over 5 runs). In particular, we compute the synthesis time when the tools are given the minimum number of examples they required to produce the correct program. Figure 8 shows the results. We restrict this experiment to FlashFill, Duet, and FlashFill++, as synthesis time for SmartFill is unobservable through the browser.

Figure 8a compares FlashFill++ with the *faster of* FlashFill and Duet. We focus on benchmarks that are solved by FlashFill++, and by either FlashFill or Duet. The y-axis displays the log base 10 of the ratio of the best of FlashFill and Duet synthesis time to the FlashFill++ synthesis time. A higher value represents a larger reduction in synthesis time. On the x-axis we display the best of FlashFill and Duet synthesis time (in milliseconds) for that benchmark task. FlashFill++ reduces synthesis time in 63% of the benchmarks, and the remaining 37% happen to be benchmarks where synthesis is fast ($< 500ms$ in most cases) and slowdown is likely not observable in a user-facing application. In 37.3% cases, FlashFill++ is at least one order of magnitude faster and in 18% cases it is at least two orders of magnitude faster. Table 8b shows the average synthesis time for various tools across the benchmark classes. We averaged over the benchmarks that the tool solved. We see that FlashFill++ has better averages despite solving more (presumably harder) benchmarks.

> FlashFill++ is faster on average despite solving more (presumably harder) benchmarks and better than the best of the baselines on most hard instances.

Finally, we evaluate the gain from using gDSLs. We create FlashFill++$^-$ by replacing gDSL used in FlashFill++ by a regular DSL ($\triangleright$ operator is treated as the usual |). We compare FlashFill++ and FlashFill++$^-$ on synthesis time and minimum examples required metrics. Figure 9b summarize the results. We note that gDSLs reduce synthesis time in 91% of the benchmark tasks, give more than 3x speedup in 20% of cases, and generate better performance across the benchmarks and metrics.

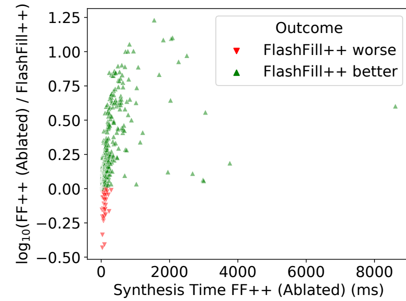
> Precedences in gDSLs consistently help FlashFill++ in improving both search (synthesis time) and ranking (minimum number of examples).

6.3 Code Readability

Traditionally DSL design has focused on efficacy of the learning and ranking process, and not on readable code generation. We evaluated the extent to which FlashFill++ enables such readable code.

	Duet	Playgol	Prose	Overall
	Average Synthesis Time			
FlashFill++	**203.0**	**217.1**	**246.4**	**228.5**
FlashFill++ $^{-}$	255.0	350.6	410.3	361.1
	Average #examples required			
FlashFill++	**1.69**	**1.46**	**1.52**	**1.53**
FlashFill++ $^{-}$	1.76	1.54	1.56	1.59

(a) Synthesis time and no. of examples required to learn.



(b) Synthesis Time Ratios.

Fig. 9. FlashFill++ improves over an ablation that removes the use of gDSLs, denoted FlashFill++$^{-}$.

First, we compared the code complexity for programs synthesized by FlashFill++ and FlashFill for two target languages: Python & PowerFx, the low/no-code language of the Power platform [PowerFx 2021]. On average, FlashFill++'s Python is 76% shorter and uses 83% fewer functions, whereas FlashFill++'s PowerFx is 57% shorter and uses 55% fewer functions. We next compared the Google Sheets formula language code generated by SmartFill with Excel code generated by FlashFill++. We found that FlashFill++ does better in ≈ 60% of the formulas generated and is at parity for ≈ 20% of the formulas. We did not compare with Duet since it generates programs only in its DSL and not in any target language.

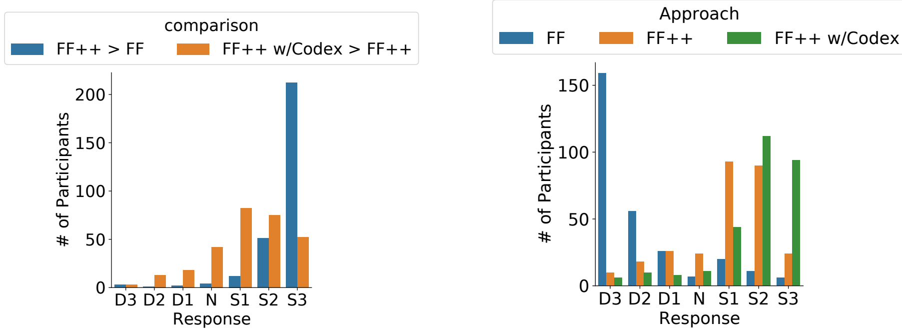
Next, we carried out a survey-based user study where we asked users to read and compare different Python functions synthesized by alternative approaches. In our study, we further augment FlashFill++ with a procedure to rename variables. This part of the system is optional, and is only added to further underscore the benefits of readable code generation. To perform variable renaming we use the few-shot learning capability of Codex [Chen et al. 2021b], a pretrained large language model, and iteratively provide the following prompt [Gao et al. 2021]: (1) two samples of the renaming task, where each sample contains I/O examples, FlashFill++ program, and the renamed program, (2) the current renaming task, which contains the examples and the FlashFill++ program to be renamed, and (3) partially renamed program up to the next non-renamed variable.

We sampled 10 tasks from our benchmarks, with probability proportional to the number of identifier renaming calls made to Codex. For each task, we displayed the Python code generated by FlashFill, FlashFill++, and FlashFill++ with Codex (anonymized as A, B, and C). For the first 5 tasks, the participants were asked (on a 7-point Likert scale) the extent to which they agreed with the statements *"FlashFill++ is more readable than FlashFill"* and *"FlashFill++ with Codex is more readable than FlashFill++"*. For the last 5 tasks, the participants answered (on a 7-point Likert scale) the extent to which they agreed with the statement *"X is similar to the code I write"*, where *X* was replaced with the corresponding (anonymized) system name.

Figure 10a shows that participants found code generated by FlashFill++ (without identifier renaming) was more readable than code generated by FlashFill for the same task. Adding Codex-based renaming further improved readability with most participants at least somewhat agreeing.

Figure 10b shows that participants strongly disagreed that FlashFill code is similar to the code they write. In contrast, most participants at least somewhat agreed that FlashFill++ code is similar to the code they write. Adding identifier renaming resulted in improvements, across all five tasks.

We also provided an open-ended text box for additional feedback with each task. Here are some illustrative excerpts, where we have replaced the anonymized system names (A,B,C) with meaningful counterparts: *FlashFill: "is a mess", FlashFill++: "very readable", FlashFill++ +Codex: "parameter name is more self describing"*; *"FlashFill is just confusing while FlashFill++ and/or FlashFill++ with Codex are*



(a) *Statement: X is more readable than Y* (denoted as $X > Y$). Most participants found FlashFill++ better than FlashFill. Adding Codex further improved it.

(b) *Statement: X is similar to the code I write.* Participants did not find FlashFill similar. FlashFill++ was closer, but FlashFill++ +Codex most similar.

Fig. 10. Survey: D3-S3 Strongly disagree/agree. FF=FlashFill, FF++=FlashFill++, FF++ w/Codex=FlashFill++ with Codex.

quite simple and direct"; and *"FlashFill is very badly written, and FlashFill++ with Codex's parameter name tells a much better story"*.

7 DISCUSSION

7.1 Related Work

Cuts are closely related to the widely-studied concept of *accelerations* in program analysis. Accelerations are used to capture the effect (transitive closure) of multiple state transitions by a single "meta transition" [Finkel 1987; Karp and Miller 1969]. It has often been used to handle numerical updates in programs, especially when more classical abstract interpretation techniques either do not converge or become very imprecise [Bardin et al. 2008; Boigelot 2003; Jeannet et al. 2014]. Our use of cuts inherits its motivation and purpose from these works, but applies them to PBE. Whereas in program analysis, accelerations had success mostly on numerical domains, in PBE we find cuts helpful more generally. Currently, we assume cuts are provided by the DSL designer, but automatically generating them remains an interesting topic for future research.

In PBE, cuts help speed-up search (by guiding top-down propagation across non-EI operators based on abstracting behavior of inner sub-DSLs). Other ways to speed-up search include using types and other forms of abstractions [Guo et al. 2020; Osera and Zdancewic 2015; Wang et al. 2018], or combining search strategies [Lee 2021]. The difference between cuts and abstraction-based methods in synthesis is the same as the difference between accelerations and abstraction in program analysis. We need cuts for only *some* operators, whereas abstract transformers are required for *all* operators. Moreover, the methods are not incompatible – a promising direction would be to combine them.

Middle-out synthesis, enabled by cuts, is a new way to combine bottom-up [Alur et al. 2013, 2017] and top-down [Gulwani 2011; Polozov and Gulwani 2015] synthesis. It is very different from the *meet-in-the-middle* way of combining them where search starts simultaneously from the bottom (enumerating subprograms that generate new values) and from the top (back propagating the output) until we find values that connect the two [Gulwani et al. 2011; Lee 2021]. Helped by the *jump* provided by cuts, middle-out synthesis starts at the middle creating two subproblems that can be solved using either approach. Meet-in-the-middle approach in [Lee 2021] guides the top-down search based on the values propagated by bottom-up, similar to middle-out synthesis; however, our

cuts are more general and can handle more scenarios because they overcome issues (not effectively enumerable operators and large number of constants) that may make partial bottom-up fail.

Middle-out synthesis can be viewed as a divide-and-conquer strategy for synthesis. The cooperative synthesis framework in [Huang et al. 2020] proposes 3 such strategies that are used when the original synthesis problem remains unsolved. However, [Huang et al. 2020] works on complete logical specifications, and not on input-output examples.

At a very abstract level, top-down synthesis and middle-out synthesis can both be viewed as an approach that synthesizes a sketch and then fills the sketch in a PBE setting [Feng et al. 2017; Polikarpova et al. 2016; Wang et al. 2017, 2020]. In this context, Scythe [Wang et al. 2017] uses overapproximation of the set of values that are computed by partial programs to synthesize a sketch. Unlike our work, [Wang et al. 2017] is exclusively focused on bottom-up synthesis. Since [Wang et al. 2017] is bottom-up, its approximations get coarser as the program gets deeper. Scythe, tends to do well for shallow programs. FlashFill++'s use of cuts is more "on-demand" and thus not affected by depth of programs. In fact, FlashFill++ can synthesize deep programs, with the largest solution in our benchmark suite having a depth of 24, with over 10% of our benchmarks requiring programs with a depth of at least 10. Furthermore, [Wang et al. 2017] is exclusively focused on SQL – its main contribution is an approach to overapproximate SQL queries that abstracts carefully selected nonterminals. In contrast, our formalization is not fixed to any particular DSL, but relies on the DSL designer to provide the cuts.

Morpheus [Feng et al. 2017] and Synquid [Polikarpova et al. 2016] overapproximate each component to prune partial programs to synthesize sketches that are subsequently filled. Morpheus is specialized to tables and uses the distinction between value and table transformations. In contrast, our framework is more general as it allows the use of approximations (cuts) for only certain functions (wherever they are provided); however, we cannot (and do not) prune partial programs. We always work on concrete values – there is no abstract domain involved. We do not use SMT solvers, whereas SMT solvers are a key component of [Feng et al. 2017; Polikarpova et al. 2016].

Precedence and gDSLs. Precedences or priorities have been used in many grammar formalisms, but mainly for achieving disambiguation while parsing in ambiguous grammars [Aasa 1995; Aho et al. 1973; Earley 1974; Heering et al. 1989]. Disambiguation here refers to preferring the parse $a+(b*c)$ over $(a+b)*c$ for the *same string* $a+b*c$. In contrast, we use gDSLs to compare derivations of *different* strings (programs). Furthermore, in the work on filters and SDF [Heering et al. 1989], the semantics of the precedence relation $\succ$ on rules is different: there $S_1 \rightarrow w_1 \succ S_2 \rightarrow w_2$ means that one can not use $S_2 \rightarrow w_2$ as a child of $S_1 \rightarrow w_1$ in a parse tree [van den Brand et al. 2002]. In our case, we disallow precedence on rules with different left-hand nonterminals. Nevertheless, our precedence can be viewed as a specialized filter in the terminology of [van den Brand et al. 2002].

Farzan and Nicolet [Farzan and Nicolet 2021] use a sub-grammar to optimize search. This can be modeled using our precedence operator. They use constraint-based synthesis (using SMT solvers) where the interest is in *any one* correct program. Ranking is not of interest there, whereas we use precedence in the context of top-down synthesis where we synthesize program sets and rank them. The interaction of precedence in the grammar and the program ranking is one of our contributions.

Casper [Ahmad and Cheung 2018] uses a hierarchy of grammars growing in size for synthesis, making search efficient. This hierarchy is dynamically generated - guided by the input-output example. Our precedence-based approach provides a different mechanism to constrain search. The value of our approach is that it is easily integrated with the underlying synthesis framework, giving synthesis-engine-builders more flexibility in controlling search and ranking. Precedence is also intuitive for DSL designers because they must think only locally to decide if the operators need a precedence relation. Technically speaking, our notion of precedence implicitly represents a lattice of grammars rather than a strict linear hierarchy.

Earlier work on "Parsing expression grammars" [Ford 2004] introduces grammars that are like CFGs, but contain features such as prioritized choice (precedence), greedy repetitions, etc., but it does so for parsing, whereas our focus is on top-down, bottom-up, and combination synthesis techniques. Our novelty is in supporting precedence in our synthesis framework, and formally working out how it impacts ranking and search.

Using precedence is one way to handle potentially redundant operators in the DSL and prune search space. The other way is to explicitly write the equivalence relation on programs and only consider programs that are canonical representatives of each equivalence class [Osera and Zdancewic 2015; Smith and Albarghouthi 2019; Udupa et al. 2013]. The gDSL approach is low overhead, but may consider equivalent programs during search. However, this is by design as our goal is to generate whatever program leads to most naturally readable code.

Precedence of grammar rules can be viewed as a specialized case of probabilistic context-free grammars (pCFGs) that have been used to bias the enumeration of programs through their grammar [Lee et al. 2018; Liang et al. 2010; Menon et al. 2013]. While specialized, precedence is actually preferable in many scenarios where we want to synthesize not just *any* program that works on the input-output examples, but one that has other desirable properties, such as, the program generalizes to unseen inputs and has a readable translation in target language. As such, precedence in gDSLs give designers of synthesizers a clean way to state their ranking preference. Weights in a pCFG have to be learned from data or manually set – both are daunting tasks compared to writing a gDSL. The contrast between pCFGs and gDSLs is akin to that between neural and symbolic approaches.

FlashFill [Gulwani 2011] demonstrated the effectiveness of inductive synthesis at tackling complex string transformation. FlashMeta [Polozov and Gulwani 2015] recognized that many popular inductive synthesizers [Gulwani 2011; Le and Gulwani 2014] could be decomposed into domain-specific features, such as the DSL operators and their semantics, and general (shareable) deductive steps. We used the FlashMeta framework to implement FlashFill, based on the original paper [Gulwani 2011], for our experiments. We also built FlashFill++ the same way, which extends the capabilities of FlashFill to include new operations like date and number formatting, and also focuses on generating readable code.

In recent work [Verbruggen et al. 2021], FlashFill has been combined with a pre-trained language model (LM), GPT3, to facilitate *semantic* transformations, such as converting the city "San Francisco" to the state "CA". Complementing FlashFill's syntactic effectiveness with GPT3's ability to do semantic transformation is interesting, but orthogonal to the problem here. However, we do exploit a LM to (optionally) generate meaningful variable names for our user study on code readability.

Trust and readability. Wrex [Drosos et al. 2020] argues that readable code is *indispensable* for users of synthesis technology. Wrex employed hand-written rewrite rules to produce readable code for their user study. However, this is not an approach that easily extends to all scenarios and larger languages. FlashFill++ is inspired by Wrex [Drosos et al. 2020] to address the readable code generation challenge in a more general and scalable way: redesigning the DSL used with a focus on enabling readable code generation, rather than post-processing.

Zhang et al [Zhang et al. 2021, 2020] introduced a system for *interpretable synthesis*, where the user interacts with the synthesizer. This approach is complementary to FlashFill++.

7.2 Limitations

The concepts of *cuts* and *precedence* have been developed exclusively for synthesis approaches based on concrete values, so-called version space algebra (VSA) based methods, in this paper. The term *top-down*, respectively *bottom-up*, has been used for techniques that are based on propagation of concrete output (respectively, input) values through partial sketches (respectively, programs) typically represented using VSAs. In systems that represent approximations of the sets of values

(at each component) using abstractions or refinement types (e.g., Synquid, Morpheus, etc), cuts can potentially address the issue of abstractions becoming too coarse as they are composed from abstractions of sub-components. This is similar to how interpolants can be used to compute tight invariants in program verification when successive application of abstract transformers lead to overly general invariants. Studying broader implications of these ideas is left for future work.

Nontrivial cuts that go beyond bottom-up or top-down approaches can only be computed if there are nontrivial invariants that hold (about the values that are generated) at certain intermediate nonterminals in the grammar. Moreover, the DSL designer must be aware of these invariants. The DSL designer can choose to write cuts that are incomplete in theory, but reasonable in practice, guided by their understanding of the domain. Nontrivial cuts are likely to exist in large DSLs where different forms of values flow on different paths. We have not done a formal study of how difficult it is for a DSL designer to write useful cut functions—this is beyond the scope of the current paper which just lays down the foundations for cuts.

Designing a guarded DSL requires the DSL designer to have clear preference for certain operators over other alternatives, and moreover, any precedence on operators closer to the start symbol (in the grammar) should override any precedence on operators closer to the leaves of the program tree. The "higher-up" operators in any DSL typically establish the high-level tactic of the program, and hence this requirement often holds. Precedences in a grammar are likely to exist if it contains redundant operators that are some preferred compositions of other low-level operators in the grammar. Further, there are certain technical assumptions we make about precedences. The assumption that the precedence is a series-parallel partial order is not required if we start with the gDSL (rather than start with precedences), which is what we do in practice. The assumption that the reachability relation on the grammar nonterminals be acyclic is required only to provide a clean mathematical interpretation of the ranking induced by gDSL on programs (terms) as a path ordering. In the presence of cycles, the synthesis rule and the full system can still be used without problems, but ranking cannot be described in a simple way.

8 CONCLUSION

We introduced two techniques, cuts and precedence through guarded DSLs, that DSL designers can use to prune search in programming by example. Cuts enable a novel synthesis strategy: middle-out synthesis. This strategy allows FlashFill++ to support synthesis tasks that require non-EI/EE operators, such as datetime and numeric transformations. The use of precedence through gDSLs allows us to increase the size of our DSL, by incorporating redundant operators, which facilitate readable code generation in different target languages. We compare our tool to existing state-of-the-art PBE systems, FlashFill, Duet, and SmartFill, on three public benchmark datasets and show that FlashFill++ can solve more tasks, in less time, and without substantially increasing the number of examples required. We also perform a survey-based study on code readability, confirming that the programs synthesized by FlashFill++ are more readable than those generated by FlashFill.

REFERENCES

Annika Aasa. 1995. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science* 142, 1 (1995), 3–26.

Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proc. 2018 International Conference on Management of Data, SIGMOD Conference*. ACM, 1205–1220. https://doi.org/10.1145/3183713.3196891

Alfred Aho, S. Johnson, and Jeffrey Ullman. 1973. Deterministic parsing of ambiguous grammars. *Commun. ACM* 18 (01 1973), 441–452. https://doi.org/10.1145/360933.360969

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Formal Methods in*

Computer-Aided Design, FMCAD 2013. 1–8.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS*. 319–336.

Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. 2008. FAST: acceleration from theory to practice. *Int. J. Softw. Tools Technol. Transf.* 10, 5 (2008), 401–424. https://doi.org/10.1007/s10009-008-0064-3

Denis Béchet, Philippe de Groote, and Christian Retoré. 1997. A Complete Axiomatisation for the Inclusion of Series-Parallel Partial Orders. In *Rewriting Techniques and Applications, 8th Int. Conf., RTA-97 (Lecture Notes in Computer Science, Vol. 1232)*. Springer, 230–240. https://doi.org/10.1007/3-540-62950-5_74

Bernard Boigelot. 2003. On iterating linear transformations over recognizable sets of integers. *Theor. Comput. Sci.* 309, 1-3 (2003), 413–468. https://doi.org/10.1016/S0304-3975(03)00314-1

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. 2021. Neurosymbolic Programming. *Found. Trends Program. Lang.* 7, 3 (2021), 158–243.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. 2021a. Spreadsheet-Coder: Formula Prediction from Semi-structured Context. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 1661–1672. https://proceedings.mlr.press/v139/chen21m.html

Andrew Cropper. 2019. Playgol: Learning Programs Through Play. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 6074–6080. https://doi.org/10.24963/ijcai.2019/841 https://github.com/andrewcropper/ijcai19-playgol.

Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 243–320.

Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew J. Hausknecht, and Pushmeet Kohli. 2017. Neural Program Meta-Induction. In *NIPS*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 2080–2088.

Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

Jay Earley. 1974. Ambiguity and Precedence in Syntax Description. *Acta Informatica* 4 (1974), 183–192.

Azadeh Farzan and Victor Nicolet. 2021. Phased synthesis of divide and conquer programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 974–986. https://doi.org/10.1145/3453483.3454089

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 422–436. https://doi.org/10.1145/3062341.3062351

Alain Finkel. 1987. A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In *Proc. 14th Intl. Colloquium on Automata, Languages and Programming, ICALP87 (Lecture Notes in Computer Science, Vol. 267)*, Thomas Ottmann (Ed.). Springer, 499–508. https://doi.org/10.1007/3-540-18088-5_43

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 111–122. https://doi.org/10.1145/964001.964011

Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making Pre-trained Language Models Better Few-shot Learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 3816–3830. https://doi.org/10.18653/v1/2021.acl-long.295

Google. 2021. SpreadSheetCoder. https://github.com/google-research/google-research/tree/master/spreadsheet_coder

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. 317–330.

Sumit Gulwani. 2016. Programming by Examples - and its applications in Data Wrangling. In *Dependable Software Systems Engineering*. 137–158.

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

S. Gulwani, V. Korthikanti, and A. Tiwari. 2011. Synthesizing geometry constructions. In *Proc. ACM Conf. on Prgm. Lang. Desgn. and Impl. PLDI*. 50–61.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL (2020), 12:1–12:28. https://doi.org/10.1145/3371080

Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. 1989. The syntax definition formalism SDF - reference manual. *ACM SIGPLAN Notices* 24, 11 (1989), 43–75.

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling enumerative and deductive program synthesis. In *Proc. 41st ACM SIGPLAN Intl. Conf. on Programming Language Design and Implementation, PLDI*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1159–1174. https://doi.org/10.1145/3385412.3386027

Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. 2014. Abstract acceleration of general linear loops. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 529–540. https://doi.org/10.1145/2535838.2535843

Ashwin Kalyan, Abhishek Mohta, Alex Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations (ICLR)* (6th international conference on learning representations (iclr) ed.). https://www.microsoft.com/en-us/research/publication/neural-guided-deductive-search-real-time-program-synthesis-examples/

Richard M. Karp and Raymond E. Miller. 1969. Parallel Program Schemata. *J. Comput. Syst. Sci.* 3, 2 (1969), 147–195. https://doi.org/10.1016/S0022-0000(69)80011-5

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *PLDI*. 542–553.

Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434335 https://github.com/wslee/duet.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 436–449. https://doi.org/10.1145/3192366.3192410

Percy Liang, Michael I. Jordan, and Dan Klein. 2010. Learning Programs: A Hierarchical Bayesian Approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 639–646.

Dylan Lukes, John Sarracino, Cora Coleman, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. Synthesis of web layouts from examples. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 651–663.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the 30th International Conference on Machine Learning, ICML (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 187–195. http://proceedings.mlr.press/v28/menon13.html

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *Proc. ACM Program. Lang.* 2, POPL (2018), 1:1–1:30.

Nagarajan Natarajan, Danny Simmons, Naren Datha, Prateek Jain, and Sumit Gulwani. 2019. Learning Natural Programs from a Few Examples in Real-Time. In *AIStats*. https://www.microsoft.com/en-us/research/publication/learning-natural-programs-from-a-few-examples-in-real-time/

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proc. 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, 619–630. https://doi.org/10.1145/2737924.2738007

Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu K. Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 785–796.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program synthesis. In *OOPSLA/SPLASH*. 107–126.

PowerFx 2021. PowerFx: The low code programming language. https://powerapps.microsoft.com/en-us/blog/introducing-microsoft-power-fx-the-low-code-programming-language-for-everyone/. Accessed: 2021-11-19.

Microsoft PROSE. 2022. PROSE public benchmark suite. https://github.com/microsoft/prose-benchmarks.

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *ICSE*. IEEE / ACM, 404–415.

Selenium. 2022. Selenium. https://github.com/SeleniumHQ/selenium

Nischal Shrestha, Titus Barik, and Chris Parnin. 2018. It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, Jácome Cunha, João Paulo Fernandes, Caitlin Kelleher, Gregor Engels, and Jorge Mendes (Eds.). IEEE Computer Society, 177–185. https://doi.org/10.1109/VLHCC.2018.8506508

Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV*. 398–414.

Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *VMCAI*, Constantin Enea and Ruzica Piskac (Eds.).

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 287–296. https://doi.org/10.1145/2491956.2462174

Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction, 11th Intl. Conf, CC 2002, Part of ETAPS, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*. Springer, 143–158.

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–25. https://doi.org/10.1145/3485477

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. *Interpretable Program Synthesis*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445646

Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900

Received 2022-07-07; accepted 2022-11-07