NLyze: Interactive Programming by Natural Language for SpreadSheet Data Analysis and Manipulation

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Mark Marron
Microsoft Research
marron@microsoft.com

ABSTRACT

Millions of computer end users need to perform tasks over tabular spreadsheet data, yet lack the programming knowledge to do such tasks automatically. This paper describes the design and implementation of a robust natural language based interface to spreadsheet programming. Our methodology involves designing a typed domain-specific language (DSL) that supports an expressive algebra of map, filter, reduce, join, and formatting capabilities at a level of abstraction appropriate for non-expert users. The key algorithmic component of our methodology is a translation algorithm for converting a natural language specification in the context of a given spreadsheet to a ranked set of likely programs in the DSL. The translation algorithm leverages the spreadsheet spatial and temporal context to assign interpretations to specifications with implicit references, and is thus robust to a variety of ways in which end users can express the same task. The translation algorithm builds over ideas from keyword programming and semantic parsing to achieve both high precision and high recall. We implemented the system as an Excel add-in called NLyze that supports a rich user interaction model including annotating the user's natural language specification and explaining the synthesized DSL programs by paraphrasing them into structured English. We collected a total of 3570 English descriptions for 40 spreadsheet tasks and our system was able to generate the intended interpretation as the top candidate for 94% (97% for the top 3) of those instances.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Program Synthesis; H.5.2 [**User Interfaces**]: Natural language

Keywords

Program Synthesis, End-user Programming, Spreadsheet Programming, User Intent

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

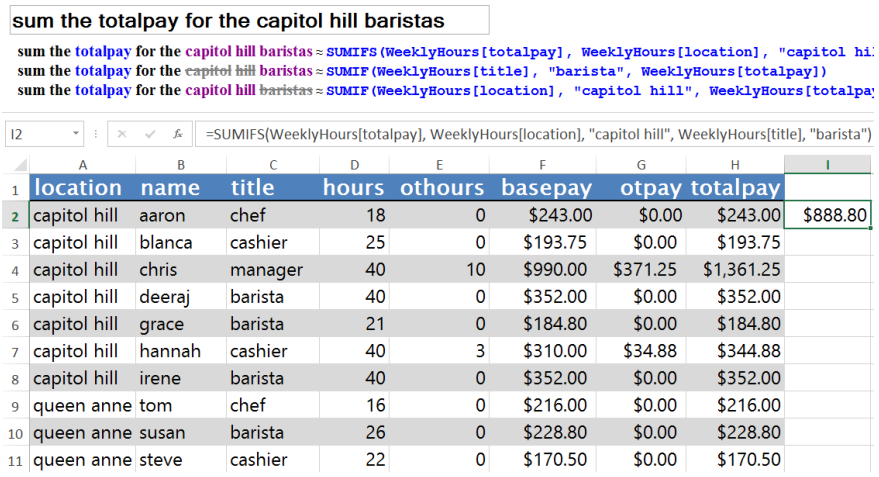
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2612177.

1. INTRODUCTION

The IT revolution over the recent decades has resulted in two significant advances: the digitization of massive amounts of data and widespread access to computational devices. It is thus not surprising that more than 500 million people worldwide use spreadsheets for storing and manipulating data. These business end users have myriad diverse backgrounds and include commodity traders, graphic designers, chemists, human resource managers, finance professionals, marketing managers, underwriters, compliance officers, and even mailroom clerks. They are not professional programmers but they often need to create small, often one-off, scripts to perform business tasks [5].

Unfortunately, the state of the art for interfacing with spreadsheets is far from satisfactory. Spreadsheet systems, like Microsoft Excel, come with a maze of features, and end users struggle to find the correct features to accomplish their tasks [16]. More significantly, programming is still required to perform tedious and repetitive data analysis/manipulation tasks. Excel allows users to write macros using a rich inbuilt library of string and numerical functions, or to write arbitrary scripts in Visual Basic or .Net programming languages. However, since end users are not proficient in programming, they find it too difficult to write desired macros/scripts. Moreover even skilled programmers might hesitate to write a script for a one-off repetitive task.

Recently, Gulwani developed a programming by example (PBE) technique to automate string manipulation tasks in spreadsheets [7] (released as the popular Flash Fill feature in Excel 2013 [1]). However, there are important classes of tasks such as reduce and filter operations that are not easily described using examples. Consider the employee payroll spreadsheet shown in Fig. 1. Suppose one wants to "sum the totalpay for the capital hill baristas". This *conditional arithmetic* task requires combination of a filter operation (namely to filter rows whose location is "capital hill" and whose title is "barista") and a reduce operation (namely to add up the totalpay values for those rows). These operations are not easily expressible using examples. In particular, filter operations would require too many examples, and reduce operations may require the user to construct small mock examples [40]. We studied Excel help forums and observed that end users often struggle with such tasks and communicate their intent to the forum experts using natural language. Inspired by this observation, we have developed a programming by natural language (PBNL) methodology and implemented a Microsoft Excel add-in called NLyze which enables end users to automate common tasks using natural language.



	A	B	C	D	E	F	G	H	I
1	location	name	title	hours	othours	basepay	otpay	totalpay	
2	capitol hill	aaron	chef	18	0	$243.00	$0.00	$243.00	$888.80
3	capitol hill	blanca	cashier	25	0	$193.75	$0.00	$193.75	
4	capitol hill	chris	manager	40	10	$990.00	$371.25	$1,361.25	
5	capitol hill	deeraj	barista	40	0	$352.00	$0.00	$352.00	
6	capitol hill	grace	barista	21	0	$184.80	$0.00	$184.80	
7	capitol hill	hannah	cashier	40	3	$310.00	$34.88	$344.88	
8	capitol hill	irene	barista	40	0	$352.00	$0.00	$352.00	
9	queen anne	tom	chef	16	0	$216.00	$0.00	$216.00	
10	queen anne	susan	barista	26	0	$228.80	$0.00	$228.80	
11	queen anne	steve	cashier	22	0	$170.50	$0.00	$170.50	

Figure 1: NLyze: Sum on Coffee Shop sheet.

Variations in Language on Same Task	Variations in Task and Composition
• sum hours capitol hill baristas	• what are the average hours worked at capitol hill
• sum the capitol hill barista hours	• for each employee lookup the payrate and multiply by hours
• sum the hours for the baristas where the location is capitol hill	• take the nonzero othours and get the hours and sum them
• computer please sum the hours for the capitol hill location baristas	• add the hours and the othours columns
• get the baristas for the capitol hill location and sum the hours	• get the rows with othours bigger than 0 and color them red
• get the hours for baristas who work at capitol hill and sum them up	• which country has the largest gdp per capita
• compute the total sum of the hours for the people who are baristas and work at the capitol hill location	• which countries have a gdp per capita larger than the average
• find sum of totalpay for all of capitol hill baristas	• sum the gdp for all countries that are not in europe
• what is totalpay sum for baristas at capitol hill	• how many countries are in europe but do not use the euro
• sum column H where column C is barista and A is capitol hill	• basepay plus otpay times 1.10
	• select rows with employees at queen anne with over 20 hours

Table 1: Variations on expressing a single intent from 67 clusters of sentences describing the same task (Left column). A selection of the variety of tasks included in the evaluation sets (Right column).

The two major challenges in this work are: (i) variability in how different users may express the same task in natural language and (ii) the range of tasks (actions and their combinations) that users want to perform. The first column in Tab. 1 shows a small selection of English descriptions that were provided by end users for the same task category. On one hand, we have a minimalistic keyword based description (as often used in search engines) such as "sum hours capital hill baristas". On the other hand, we have a verbose description such as "computer please sum hours for the capitol hill location baristas". There are also descriptions that use implicit references (e.g., "get the hours for the baristas who work at capitol hill and sum *them*") or use linguistic idioms or implicit relations in the table structure (e.g., "capitol hill baristas" instead of explicitly stating the "and" conjunction, or simply saying "capitol hill" instead of the explicit "location column equals capitol hill").

Besides observing linguistic variety in task descriptions, we also observed that users want to perform a rich composition of some basic functions. The second column in Tab. 1 shows a sample of the kinds of actions and their combinations that we encountered in our evaluation and user study. There are task descriptions that involve multiple Boolean connectives including negation (e.g., "how many countries are in europe but do not use the euro"), that nest reduction operations inside other functions (e.g., "which countries have a gdp per capita larger than the average"), and that combine lookup functions with other operations (e.g., "for each employee lookup the payrate and multiply by hours").

The first step in our methodology is to design a domain-specific language (DSL) that on one hand is expressive enough to express desired categories of tasks. On the other hand it should be restricted enough to allow effective translation from varied descriptions in natural language and to allow for a simple end-user friendly interaction model. With these goals in mind, we developed a richly-typed DSL that supports compositions of some basic forms of map, filter, and reduce operations. Its typed and compositional nature enables effective translation from natural language. Its compositional nature and the choice of core operators allow for automation of a wide variety of common spreadsheet tasks that end users struggle with (such as *conditional filtering*, *conditional arithmetic*, and *lookup* tasks). The DSL also supports selecting and formatting of spreadsheet cells. This not only supports common *conditional formatting* tasks, but also allows users to define views that can be referenced in subsequent computations, thereby allowing users to perform complex tasks in a sequence of smaller steps.

The key algorithmic contribution of this paper is a translation algorithm for converting natural language specifications into a ranked set of likely programs in our DSL. Our translation algorithm improves over and combines ideas from two different approaches, namely keyword programming and semantic parsing, that have been used in the literature in different communities. The idea of keyword programming, which has been explored in the Programming Languages (PL) community [26, 31, 39] and the Human-Computer Interaction (HCI) community [24, 25], is to generate all valid

programs that can be obtained by combinations of user provided tokens or their representative keywords. Keyword programming approaches have high recall but low precision and rely on the user to select the intended program from among several options. The idea of semantic parsing, which has been explored in the Natural Language Processing (NLP) community [28, 32, 42] and the Database (DB) community [11, 21, 33], is to use a set of rules to translate well-formed sentences into their corresponding logical representations. Semantic parsing approaches typically have high precision but lower recall and can be sensitive to grammatically incorrect or ill-formed specifications.

Our translation algorithm uses a dynamic programming based approach and an effective ranking scheme to combines ideas from keyword programming and semantic parsing. Our translation algorithm employs a more sophisticated form of template rules than have traditionally been used for semantic parsing. It also leverages the spreadsheet spatial and temporal context to assign interpretations to natural language specifications with implicit references, and is thus robust to a variety of ways in which end users can express the same task. It thus achieves both high precision and high recall on a challenging benchmark consisting of real world colloquial descriptions of spreadsheet tasks.

This paper makes the following contributions:

- We present the design of a richly-typed DSL for spreadsheet programming (§2). Our language supports a compositional algebra of basic map, filter, and reduce operations and provides first class treatment to highlighting/formatting of spreadsheet cells and their referencing. These features allow natural expression of a wide variety of spreadsheet tasks that end users struggle with and also enable effective translation from natural language specifications and a simple user interface.
- We present a novel translation algorithm for translating natural language specifications into a ranked set of programs in the DSL (§3). Our translation algorithm combines ideas from keyword programming and semantic parsing and leverages spreadsheet context to interpret specifications with implicit references.
- We describe an interactive data programming environment (§4) around our DSL and translation algorithm. This includes (a) ambiguity resolution by annotating user's specification and paraphrasing of synthesized DSL programs, and (b) programming of sophisticated tasks in steps by issuing a sequence of DSL programs that communicate through spreadsheet updates.
- We collected a large real world data set and performed an extensive evaluation of the robustness of our translation algorithm to a variety of tasks and to a variety of ways in which users may express the same task (§5). We also performed controlled experiments to evaluate the various components of our translation algorithm.

2. DOMAIN-SPECIFIC LANGUAGE

Fig. 2 describes our command style DSL for programming spreadsheets. The DSL is structured around a core algebra of reduce, filter, and map operations (partly inspired by SQL) and their type-safe compositions. This allows support for *conditional arithmetic*, *conditional formatting* and *lookup* operations, and their composition. These are key categories

Program	:=	$\texttt{MakeActive}(Q) \mid \texttt{Format}(fe, Q) \mid v \mid V$
Query Expr Q	:=	$\texttt{SelectRows}(rs, f) \mid \texttt{SelectCells}(\tilde{C}, rs, f)$
Row Source Expr rs	:=	$\texttt{GetTable}(Tbl) \mid \texttt{MTable}()$
	|	$\texttt{GetActive}() \mid \texttt{GetFormat}(Tbl, fe)$
Format Expr fe	:=	$\{fmt_1, \ldots, fmt_n\}$
Filter Expr f	:=	$relop(C, v) \mid relop(v, C) \mid relop(C, C)$
	|	$\texttt{And}(f, f) \mid \texttt{Or}(f, f) \mid \texttt{Not}(f) \mid \texttt{True}$
Scalar Expr v	:=	$rop(C, rs, f) \mid \texttt{Count}(rs, f) \mid bop(v, v)$
	|	$\texttt{Lookup}(v, rs, C, C) \mid c$
Vector Expr V	:=	$bop(V, V) \mid bop(V, v) \mid bop(v, V) \mid C$
	|	$\texttt{Lookup}(C, rs, C, C)$
Format Fn fmt	:=	$\texttt{Color}(c) \mid \texttt{FontSize}(c) \mid \texttt{Bold}(b)$
	|	$\texttt{Italics}(b) \mid \texttt{Underline}(b) \mid \ldots$
Binary Fn bop	:=	$\texttt{Add} \mid \texttt{Sub} \mid \texttt{Mult} \mid \texttt{Div}$
Reduce Fn rop	:=	$\texttt{Sum} \mid \texttt{Avg} \mid \texttt{Min} \mid \texttt{Max}$
Relational Fn $relop$	:=	$\texttt{Lt} \mid \texttt{Gt} \mid \texttt{Eq}$

Figure 2: DSL: C denotes a column name. T denotes a table name. c denotes a scalar constant, while b denotes a Boolean constant.

of tasks that spreadsheet users struggle with, as we observed on Excel help forums. Furthermore, since users solicit help at this level, this seems to be the right level of abstraction at which users think or plan out their overall tasks.

A program in the DSL reads and updates the underlying spreadsheet over which it is executed. We model a spreadsheet as a collection of tables Tbl, where each table is a set of rows and has uniquely labeled and typed columns C. Each spreadsheet cell has formatting attributes including Boolean attributes like `Bold` and `Underline` and quantitative attributes like `Color` and `FontSize`. A program either produces a scalar value v or a vector value V (which is placed at the location of the active cursor in the spreadsheet), or results in highlighting ($\texttt{MakeActive}(Q)$) or formatting ($\texttt{Format}(fe, Q)$) of a set of spreadsheet rows/cells filtered by the query expression Q. This design enables the steps programming model that we discuss later in §4. Fig. 1 shows the result of a program that performs a reduce `Sum` operation and whose result is placed in the active cell `I2`.

A query expression Q returns a set of spreadsheet rows or cells. The query expression $\texttt{SelectCells}(\tilde{C}, rs, f)$ takes as input a set of columns $\tilde{C}$, a row source expression rs, and a filter expression f, can be likened to the standard SQL select-from-where expression. It filters the set of rows identified by rs using f and then projects the result to the columns $\tilde{C}$. The query expression $\texttt{SelectRows}(rs, f)$ selects the entire rows (i.e., all the columns) that are filtered by f in rs. In both cases the result set of cells/rows is activated to enable further processing via natural language or via interactive manipulation such as copy/paste or formatting.

A row source expression rs returns a set of spreadsheet rows in one of three ways. The construct $\texttt{GetTable}(T)$ returns the set of all rows in spreadsheet table T. (For readability, we drop the argument T in our examples whenever there is a single table or the context makes it clear.) The $\texttt{GetActive}()$ construct returns the set of all rows that contain the active cells in the spreadsheet. The $\texttt{GetFormat}(T, fe)$ construct returns the set of all rows in the given table that contain cells whose attributes match the collection of formatting attributes (such as $\texttt{Bold}(true)$, $\texttt{Color}(pink)$).

A filter expression f maps a row to true or false. It is a Boolean expression that includes standard Boolean connectives and whose predicates involve standard relational functions, such as `Eq` and `Lt`, over values in specific columns.

A scalar value can be produced by performing standard operations on scalar values, or by using a reduce operator *rop* (e.g., summation, average, min, max), or by using `Count` or `Lookup`. The construct $rop(C, rs, f)$ takes as input a column name C, a row source expression rs, and a filter expression f. It performs a conditional arithmetic computation by first filtering the set of rows in rs using f and then applying the reduce function *rop* to values in column C. The Count construct $\texttt{Count}(rs, f)$ takes as input a row source expression rs and a filter expression f. It returns the number of rows in rs that satisfy f. The construct $\texttt{Lookup}(v, rs, C_1, C_2)$ takes as input a value v, a row source expression rs, a primary key column name C_1 and another column name C_2. It returns the value in C_2 for that row whose value in C_1 is equal to v.

A vector value can be produced by either referring to a column C or by performing a map operation in one of three ways: (a) by applying a scalar binary function *bop* pairwise on the elements from two vectors of the same size as in $bop(V, V)$, (b) by applying the scalar binary function *bop* to a vector V and a scalar v as in $bop(V, v)$ or $bop(v, V)$, to each element in the input vector, and (c) by using a lookup construct $\texttt{Lookup}(C_1, rs, C_2, C_3)$, which returns a vector whose i^{th} element is equal to $\texttt{Lookup}(v_i, rs, C_2, C_3)$, where v_i denotes the i^{th} element in C_1. The vector valued version of the lookup function allows users to perform the equivalent of a *single column join*, based on the primary key column from the second table, with the data in the current table.

The DSL supports a strict, but intuitive, type system whose formal description is left out for space constraints. For example, multiplication is well defined on two numbers, or a number and a currency, but not on two currency values [12]. The vector operations are defined only on vectors of the same size. Each reference to a column name should be consistent with the table in scope. We encapsulate these constraints using the function `Valid`, which takes as input a DSL expression e and returns true iff e is well-typed. The translation algorithm (in §3) makes use of this function.

3. TRANSLATION ALGORITHM

We now describe our algorithm (Algo. 1) for translating a user's natural language input into a ranked set of likely programs in our DSL. We first describe the main algorithm and then describe the sub-algorithms in subsequent subsections.

Algo. 1 takes as input an English sentence S and a spreadsheet H, and returns an ordered list of top-level expressions (i.e., programs) in our DSL that are likely interpretations of the sentence S over spreadsheet H. The algorithm is based on dynamic programming and it iteratively computes the set of all expressions that can be produced for larger and larger contiguous sub-sequences (also referred to as *fragments*) of the sentence S (Loop at line 1). It then returns the set of expressions ordered by score (as defined on Line 8).

The iterative computation of expressions in the loop at line 1 is performed using two different kinds of algorithms, namely *Synth* (Algo. 2) and *Rule* (Algo. 3). Both these algorithms take as input a sentence S, indices (or word positions) i and j into the sentence and a spreadsheet H, and return a set of expressions (in our DSL) that are likely interpretations of $S[i..j]$ over H. The type-based synthesis algorithm *Synth*

Algorithm 1: *Translate*

```
   Input: Sentence S, Spreadsheet H
1  for span ← 1; span < S.Len; ++span do
2      for pos ← 0; pos < S.Len − span; ++pos do
3          end ← pos + span;
4          rulev ← Rule(S, pos, end, H);
5          synthv ← Synth(S, pos, end);
6          TMap[pos, end] ← rulev ∪ synthv;
7  for e in TMap[0, S.Length − 1] do
8      pscore(e) ← ProdSc(e) × CoverSc(e) × MixSc(e);
9  return expressions in TMap[0, S.Len − 1] ordered by pscore;
```

does this by generating all type-safe compositions (Algo. 2, line 5) of expressions that are (recursively) generated from smaller fragments (Algo. 2, line 2). The rule based translation algorithm *Rule* does this by applying a set of pattern rules (Algo. 3, Loop at line 2) which build up expressions based on matching words in the user input and the sets of previously computed expressions (Algo. 3, line 16).

The pattern rule based translation algorithm (Algo. 3 in §3.3) relies on matching common patterns and idioms in natural language to construct the appropriate expression. This type of algorithm has high precision but suffers from low recall when the natural language input is outside of the set of expected patterns. To enable translation of inputs that do not nicely match these expected patterns we introduce a second type of translation algorithm based on type driven expression synthesis (Algo. 2 in §3.2) that trades precision for higher recall by ignoring much of the structure of the user's input. Applying the combination of these algorithms at each step in a bottom-up dynamic programming manner (in Algo. 1) combines the complimentary strengths of the two translation algorithms. The high precision pattern rules are applied as much as possible while the type based synthesis is used whenever needed to synthesize expressions for parts of the input that are outside of the set of common/known patterns. The ranking, described in §3.4, selects the most likely results based on the sequence of applied operations and other features of the result expression.

3.1 Preliminaries

Partial Expression. A DSL expression e is either an atom (a numeric/currency value, a column name, or a value from the sheet), or a function/operator applied to a list of arguments. A partial expression extends the notion of an expression to also allow for a *hole*, a symbolic placeholder for an expression, as an argument.

$$\texttt{Hole} ::= \square\phi i \text{ where } i \in \mathbb{N} \wedge \phi \in \{\texttt{G}, \texttt{L}, \texttt{C}, \texttt{V}\}$$

A `Hole` has an integer identifier in $\mathbb{N}$ which can be used to refer to it and a restriction symbol ϕ that provides restrictions on what kind of expression can be used to instantiate the hole. The `G` value indicates a general hole with no restrictions on the expression that fills it, `L`, `C`, and `V` values indicate the expression should be a literal, column header, or a sheet value respectively. In order to avoid verbosity, we often refer to a *partial expression* as simply an *expression*.

As an example the `Add` operator from the DSL definition in Fig. 2 can be fully instantiated to produce expressions such as `Add(3, 5)` or `Add(3, Sub(8, 3))`. We can also partially instantiate the operator, assigning a hole to the second argument, to produce `Add(3, □G1)`. To ensure that the first argument can only be bound with a literal value we can instead partially instantiate it as `Add(□L2, □G1)`.

Given an expression $e = F(\ldots, \Box\phi i, \ldots)$ with a hole $\Box\phi i$, we can substitute another expression e' to fill the hole:

$$e[\Box\phi i \leftarrow e'] = \begin{cases} \{F(\ldots, e', \ldots)\} & \text{if } \Delta \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta \equiv e' \text{ is consistent with } \phi \wedge \texttt{Valid}(F(\ldots, e', \ldots))$$

This can be generalized to substitute multiple values at the same time, denoted as $e[\Box\phi_m m \leftarrow e_m, \ldots, \Box\phi_n n \leftarrow e_n]$.

Derivation History. A derivation history for an expression e consists of the following fields.

$$\mathit{UsedW}(e) = \text{Positions of words used to generate } e$$
$$\mathit{UsedCW}(e) = \text{Positions of column names used to generate } e$$
$$\mathit{History}(e) = (\mathit{rule}, [e^r_1, \ldots, e^r_j], [e^s_1, \ldots, e^s_k])$$

UsedW is the set of the positions of all those words that were matched by rules that instantiated e and any sub-expressions of e. *UsedCW* consists of the positions of the words that were matched to produce column names that appear in e or any sub-expressions. *History*, is a 3-ary tuple consisting of the top-most pattern rule that instantiated e, the list of expressions bound during this instantiation, and the list of expressions that were used during any synthesis steps.

3.2 Synthesis Based Translation Algorithm

The synthesis based translation algorithm (Algo. 2) computes the closure of a set of expressions (taken from the results memoized in *TMap*) by enumerating all possible well-typed combinations of the expressions in the set. In this respect, it ignores any sentence structure aside from the keywords matched to produce the initial expression set.

Combination. The combination operator *CombAll* generates the set of all expressions that can be obtained by substituting a given expression e' in some hole of another given expression e. Given expressions $e = F(e_1, \ldots, e_k)$ and e':

$$\mathit{CombAll}(e, e') = \begin{cases} \bigcup_{1 \le i \le k} \mathit{Subs}(e_i), \text{if } \mathit{Used}_{NC} \cap \mathit{Used}'_{NC} = \emptyset \\ \emptyset \qquad \text{otherwise} \end{cases}$$

$$\text{where} \quad \mathit{Used}_{NC} \equiv \mathit{UsedW}(e) - \mathit{UsedCW}(e)$$
$$\mathit{Used}'_{NC} \equiv \mathit{UsedW}(e') - \mathit{UsedCW}(e')$$

$$\mathit{Subs}(e_i) = \begin{cases} e[\Box\phi j \leftarrow e'] & \text{if } e_i \equiv \Box\phi j \\ \{F(\ldots, e''_i, \ldots) | e''_i \in \mathit{CombAll}(e_i, e')\} & \text{otherwise} \end{cases}$$

Combination proceeds by (1) iterating over all arguments of e and producing substitutions for positions with holes that are *compatible* with e' and (2) recursively performing the substitution in sub-expressions. The condition $\mathit{Used}_{NC} \cap \mathit{Used}'_{NC} = \emptyset$ checks that the two expressions do not use overlapping sets of words from the user input (aside from words used to produce column expressions). This ensures that the set of possible combinations has a finite bound based on the number of words in the input. The needed validity checks, such as type safety and that the appropriate column names exist in the table, occur during `Valid` check in the substitutions, $e[\Box\phi j \leftarrow e']$, and ensures that all result expressions are well defined wrt. the DSL semantics.

Synthesis Example. Consider the scenario when the value for *result* on line 2 in Algo. 2 is:

$$\{\$10, 5, \mathrm{Lt}(\Box\mathrm{L1}, \mathrm{totalpay}), \mathrm{Not}(\Box\mathrm{G2})\}$$

Algorithm 2: *Synth*

```
   Input: Sentence S, int i, int j
1  if (i = j) then return ∅;
2  result ← TMap[i, j − 1] ∪ TMap[i + 1, j];
3  repeat
4  |  oldResult ← result;
5  |  new ← ⋃_{e1,e2 ∈ result} {CombAll(e1, e2)};
6  |  result ← result ∪ new;
7  until oldResult = result;
8  return result;
```

There are multiple combinations possible for this set. One choice is to combine 5 and Lt(□L1, totalpay) to produce Lt(5, totalpay). We could also combine \$10 and Lt(□L1, totalpay) to get Lt(\$10, totalpay). At this point the synthesis algorithm uses the type information from the Excel document. As we treat currency and integers as different types, only one of the combinations is `Valid`. In our example the type of the "totalpay" column is a `currency` and thus only Lt(\$10, totalpay) is produced. In a later iteration the algorithm can combine the Lt expression with the hole in Not(□G2) to produce Not(Lt(\$10, totalpay)) as well.

3.3 Rule Based Translation Algorithm

The rule based translation algorithm (Algo. 3) applies a set of *rules* `Rules` to translate the fragment $S[i..j]$ into a set of expressions. A rule $T \rightarrow e$ comprises of a *template* T and a partial expression e. To support ranking we associate a numeric score ($\mathit{score} \in [0, 1]$) with each rule. Each rule $T \rightarrow e$ in `Rules` generates a set of expressions (indicated by E at Line 18) by *aligning* the template T against the fragment $S[i..j]$ (Line 3) and by filling in some (or all) of the holes in the expression e appropriately. The latter process may make use of the translations for smaller fragments that are in *TMap* (Line 16). We next describe these aspects in detail. §3.3.1 describes the format of rules and how they are learned from some training data. §3.3.2 describes the translation algorithm that makes use of these rules.

3.3.1 Rules

Rule Language. Fig. 3 describes the rule language. A *rule* consists of a template, a partial expression, and a score. A *template* is a sequence of *patterns*. There are various kinds of patterns, which we explain below. The `MustPat` pattern uses multi-word sequences for matching fragments that are fundamental to understanding the central intent of a user description. Each option consists of a multi-word phrase and we require exactly one of these to be present in the alignment. The `OptPat` pattern provides a flexible mechanism for matching (multiple) words that may optionally appear and provide extra context for the user description. To enable the rules to exclude any sheet-concept specific words and provide additional recall, the `OptPat` alignments can include an optional slack word (`SLACK`) that provides an ability to drop words that may be specific to a given spreadsheet or domain (and would not be part of a general purpose rule set).

The next patterns match numeric or currency literals as well as columns and non-numeric spreadsheet values, such as "chef" or "capitol hill". The pattern `LiteralPat` matches any literal or cell reference (e.g. `D2`) that contains a number or currency value. The integer identifier is a unique label used to associate the value matched by the pattern with the location in rule expression where the value should be placed.

`Rule`	:=	`Template` → `Expression`
`Template`	:=	`Pattern`$_1$ ... `Pattern`$_j$
`Pattern`	:=	`MustPat` | `OptPat` | `LiteralPat`
	|	`ValuePat` | `SpanPat`
`MustPat`	:=	$(w_{11} \ldots w_{1k} \mid \ldots \mid w_{i1} \ldots w_{ij})$
`OptPat`	:=	$(w_m \mid \ldots \mid w_n)^*$`SLACK?`$(w_m \mid \ldots \mid w_n)^*$
`LiteralPat`	:=	%Li where $i \in \mathbb{N}$
`ValuePat`	:=	%Ci|%Vi where $i \in \mathbb{N}$
`SpanPat`	:=	%i where $i \in \mathbb{N}$

Figure 3: The language for rules.

User descriptions, such as those in Tab. 1, are executed in the context of a spreadsheet, such as Fig. 1, which provides meaning to column name references, like *hours*, and to special value names, like *baristas*, as well as to other tables and the columns defined in them, like the *PayRates* table which contains the *payrate* column. The `ValuePat` matches column names (e.g., "title") with %Ci or spreadsheet values (e.g., "chef") with %Vi. Similar to the literal patterns the `ValuePat` matches have integer identifiers to associate the matched pattern with the location in rule expression where the value should be placed.

The final pattern for the rules is a general `SpanPat` pattern. It matches a non-deterministic span of words in the user input and is roughly equivalent to the non-deterministic regular expression "(\w+\s)+". Again, we associate a unique identifier with each span pattern that appears in a rule.

The span pattern does not place any structural restrictions on the range of matched words. This choice is critical to the feasibility of Algo. 1, as it decouples the algorithm that computes the expression semantics for the span of words from the algorithm that uses the results. Thus, we can dynamically decide to apply different (or multiple) translation algorithms on various subcomponents of the sentence depending on the words in the span. For example we could easily extend the Algo. 1 to include a parser for Excel formula to allow for a mixture of NL and Excel formula in the input, e.g. "highlight rows with totalpay > MEDIAN(H2:H14)". Further, due to the uninterpreted nature of the holes [29], we do not need to modify (or re-train) the existing *Rule* or *Synth* algorithms when adding the Excel parsing algorithm!

Learning Candidate Rules. In order to learn the various rules and their scores, we utilize a variation on the technique described by Kate et.al. in [14]. For each of the partial expressions for which we want to learn rules, we gather all examples from our training set that contain the desired subexpression. To focus the learning on a subexpression of interest we first delete, or replace by holes, the unrelated parts in the overall expressions. Simultaneously we heuristically delete the corresponding components of the NL input. We then cluster the inputs based on order and word similarity. For each cluster we derive one or more rules.

Learning Candidate Rule Example. Consider the case below where we have an example of an input along with the desired expression that matches the target expression for which we want to learn a rule:

Input = "sum the totalpay where hours less than 20"
Desired = Sum(totalpay, Lt(hours, 20)))
Target = Sum(□C1, □G2)

The first step is to unify the *desired* expression with the *target* expression to determine which subexpressions are mapped to holes. In our example this produces the mappings □C1 → totalpay and □G2 → Lt(hours, 20). With this mapping we use synonym sets derived from wordnet [38] to identify the likely set of English words associated with these expressions. In our example the sets are totalpay → {totalpay}, hours → {hours}, 20 → {20, twenty}, Lt → {less, smaller, under, ...}. We then delete words in these sets from the input sentence, replacing the deleted segments with the spans from the mappings, to get: "sum the %C1 where %2".

After performing this extraction on all of the examples that unify with the target expression we cluster the rules derived from the specific examples into equivalence classes based on structure and word similarity. In our example the sentences "sum all %C1 where %2" and "sum %C1 %2" both end up in the same cluster. We then unify all the sentences in a cluster, making words associated with the target expression *must* patterns and the remaining words *may* patterns. The result of this unification is the rule:

sum (all|the)* %C1 (where)* %2 → Sum(□C1, □G2)

Rule Selection and Scoring. Once the candidate learning task has been performed for each partial expression, we end up with an over-approximation of the desired rule set due to multiple non-deterministic choices in mapping holes and due to multiple possible rules for each cluster. To prune the rule set we iteratively run the translation algorithm with the current rule set over our training data. For each rule r, we compute a goodness score $goodness(r) = pos(r)^2/(pos(r) + neg(r))$ where $pos(r)$ is the number of examples that were translated correctly (i.e., the desired expression was one of the expressions produced by the algorithm) and where the rule was applied, and $neg(r)$ is the number of examples that were not translated correctly and where the rule was applied. For each rule we also compute the set of examples for which the rule was applicable. The goodness score reflects the overall accuracy and coverage of the rule. We then attempt to discard any rules with low goodness scores or those that are subsumed by another rule with more general applicability and with an equivalent, or better, goodness score. This process is repeated until we cannot remove any more rules without reducing the F1 score on the training data. At this point we stop and use a *Naive Bayes Classifier* to estimate the score to assign to each rule.

Rule Selection Example. After a set of rules is produced there may be multiple rules, some more useful than others, that capture similar concepts. Consider the set:

sum → Sum(□C1, Lt(□C2, □L3))
sum (the)* %C1 %C2 less %L3 → Sum(□C1, Lt(□C2, □L3))
sum (all|the)* %C1 → Sum(□C1, True)
sum (all|the)* %C1 %2 → Sum(□C1, □G2)

In this example the first rule is very general, matching any sentence containing the word "sum", but produces a very specific expression which is unlikely to be what the user intended; so it will have a low goodness score and will be discarded. The remaining rules all have high goodness scores but the second rule is subsumed by the last rule (i.e., the last rule is more general) and eliminating the second rule from the rule set does not have any impact on precision/recall on the training set. So the second rule is discarded as well. However, the third rule applies in some important cases

where the last rule does not, e.g., "sum the hours" which has no predicate, and thus eliminating it drops the precision/recall substantially; so it is retained in the set.

Unbound Holes. A rule may have holes in the expression with labels that do not correspond to any patterns in the template. These holes are left *unbound* by the rule based algorithm and are available for substitution by the synthesis algorithm (Algo. 2). An example of such a rule is:

$$\text{sum (the|values)}^* \ \%\text{C1} \rightarrow \text{Sum}(\square\text{C1}, \square\text{G2})$$

In this rule the hole defined for the column header has a pattern with the corresponding identifier on the left-hand side and it will be instantiated during Algo. 3. However, the *general* hole with identifier 2 does not have a corresponding pattern; so it will not be instantiated and is available for substitution later in Algo. 2.

Alignment. Given a template $T = \texttt{pat}_1 \ldots \texttt{pat}_j$ and a fragment $S = w_1 \ldots w_k$, we define the set of valid alignments of the template T and the fragment S as:

$$\begin{aligned}&\text{Align}(S, T) = \text{set of all alignment mappings } A \text{ where:}\\ &\quad A : \{\ \text{pat}_1, \ldots, \text{pat}_j\} \mapsto \{[l, u] | 1 \leq l \leq u \leq k\}\\ &\qquad \wedge \forall\ \text{pat}_i\ \text{Match}(\text{pat}_i, S[A(\text{pat}_i).l, A(\text{pat}_i).u])\\ &\qquad \wedge A(\text{pat}_1).l = 1 \wedge A(\text{pat}_j).u = k\\ &\qquad \wedge \forall\ \text{pat}_i, \text{pat}_{i+1} \Rightarrow A(\text{pat}_i).u + 1 = A(\text{pat}_{i+1}).l\end{aligned}$$

An alignment is a map from each of the patterns to a range of words in the sentence. The first condition ensures that each pattern is mapped to an appropriate range of words in the sentence. The next two conditions ensure that all patterns together cover the entire sentence and there are no gaps/overlaps between the ranges covered by each pattern.

3.3.2 Translation Using Rules

Algo. 3 takes a span of words $S[i \ldots j]$, finds all possible alignments of the rule templates on this span (line 3), and then produces all possible substitutions of the partial expression in the rule based on the alignments (lines 5-18). Filling holes in an expression in a rule involves looping over each hole that appears in the expression. In the case where there is no pattern with a corresponding identifier in the template T the hole is skipped and left unbound (Line 6). Otherwise the next step is to use the identifier i from the hole to lookup the corresponding pattern from the template to resolve the lower l and upper u bounds for the pattern match in the alignment. This is done using the function *LookupRangeForPatternID* (Line 8).

Once the span is known the algorithm proceeds to the switch statement in the rule application algorithm and does a case split on the given hole restriction to produce the set of expressions that can be used to fill the hole. In the case of literal (`L`) holes the words in the corresponding range are converted into the equivalent literal expression values. For the sheet value (`V`) restriction the words in the range are compared with the values seen in the spreadsheet H and the expressions corresponding to these values are returned. For the column header case the *ResolveCol* function must determine if the matched text represents a column name or if the matched text is a sheet value. If the text represents a column name in H then it can be converted directly. In the case where the text represents a sheet value but the corresponding hole restriction is `C` then the columns that contain the value in the spreadsheet H must be identified and this set of column header expressions is returned. The final case is the general restriction (`G`) which can be instantiated with any expression. To compute this set the given range is looked up in the *TMap* table.

Algorithm 3: *Rule*

```
   Input: Sentence S, int i, int j, Spreadsheet H
1  result ← ∅;
2  foreach T → e ∈ Rules do
3      foreach A ∈ Align(S[i..j], T) do
4          B ← ∅;
5          foreach □φx ∈ e do
6              if □φx is unbound in T then
7                  Continue;
8              [l, u] = LookupRangeForPatternID(A, T, x);
9              switch φ_i do
10                 case L:
11                     B[□φx] ← MakeLiteral(S[l, u])
12                 case V:
13                     B[□φx] ← MakeValue(S[l, u], H)
14                 case C:
15                     B[□φx] ← ResolveCol(S[l, u], H)
16                 case G:
17                     B[□φx] ← TMap[l, u]
18         E ← {e[□φm ← e_m, ..., □φn ← e_n] | e_k ∈ B[□φk]};
19         result ← result ∪ E;
20 return result;
```

Pattern Rule Application Example. We now illustrate Algo. 3 using the following input and two rules:

Input: "sum the totalpay for the chef titles"
Rules: sum (all|the)* %C1 %2 → Sum(□C1, □G2)
%V1 %C2 → Eq(□V1, □C2)

One possible alignment for this input maps %C1 ← "totalpay" and %2 ← "for the chef titles". There is only one possible column expression for the column header "totalpay" and this column expression is substituted into the hole □C1. There are several expressions that could have been previously computed for the sentence fragment "for the chef titles" including `Eq(chef, title)` and the expression `title`. However, only `Eq(chef, title)` has a type of `Filter` that matches the type signature of `Sum` in the DSL. Thus, the only possible result is `Sum(totalpay, Eq(chef, title))`.

3.4 Ranking

The multiple DSL expressions produced by the translation algorithm are ranked using the product (Algo. 1, line 8) of the scores, which we describe below.

Production Score. The first feature is based on the way in which an expression was produced in terms of the rules used and the number of times a hole was filled using a synthesis operation vs. pattern rule application.

$$\textit{ProdSc}(e) = \sum_{e' \in \textit{SubExprs}(e)} \frac{\textit{RScore}(e') * \textit{SScore}(e')}{|\textit{SubExprs}(e)|}$$

To compute the overall score we take the product of *RScore* and *SScore* values for all non-terminal sub-expressions (including the top-level expression) *SubExprs* and normalize their sum by the total number of non-terminal sub-expressions.

$$RScore(e) = \sum_{e^r_j \in \{e^r_1,\ldots,e^r_m\}} \frac{rule.score + History(e^r_j).rule.score}{2m}$$

$$SScore(e) = \prod_{e^s_k \in \{e^s_1,\ldots,e^s_n\}} rule.score * History(e^s_k).rule.score$$

where $History(e) = (rule, [e^r_1, \ldots, e^r_m], [e^s_1, \ldots, e^s_n])$

Intuitively, we want to favor the application of rules with high scores and prefer expressions constructed via pattern rule applications over the use of synthesis combinations. We scale the score of the rules to the range $[0, 1]$ by using the average operator to combine score for sub-expressions in the pattern rules and multiplication operator for the synthesis applications. Repeated applications of synthesis quickly drives the production score to 0 while repeated applications of pattern rules will slowly converge towards an average of the rule scores in the expressions. During the scoring of the pattern rules we sum the pairwise average of the score for the rule that was applied and the scores of each sub-expression that was bound to the rule. However, for the synthesis rules, which we have lower confidence in, we take the product of the scores for the sub-expressions in the synthesis steps.

Coverage and Order Scores. Algo. 1 may produce expressions that ignore or reorder parts of the user input. In some cases these actions are needed to make sense of the user input, but they may also lead to erroneous expressions.

The coverage score *CoverSc* ranks the expression based on how completely it covers the words in the user input. The intuition for this is that if a user included a word in the input then this word conveys some information on the user's intent. An expression that covers a larger number of words is seen as better explaining the input and thus having a higher likelihood of correctly capturing the intent behind it. Our formulation is non-linear to strongly down weight expressions that ignore large numbers of words, presumably some of which are important to the user's intent, while not unduly penalizing expressions that ignore a few possibly redundant words. Thus, given the input $S = w_1 \ldots w_m$ and the expression $e = F(e_1, \ldots, e_k)$ we define:

$$CoverSc(e) = 1/Max((m - UsedW(e))^2, 1)$$

The mix score *MixSc* recursively counts all pairs of sub-expressions that mix words from different parts of the user input. Intuitively we want to allow the re-ordering of parts of the user input but we should not mix different fragments of the input. For example in a conditional sum like "get the rows where othours is less than 20 and sum the hours", the condition/reduction may be expressed at the beginning or end of the sentence, which we want to allow the synthesizer to reorder, but we do not want to swap the column name that appears in the condition, "othours" with the column name that appears with the sum "hours".

$$MixSc(e) = \begin{cases} 1 - Swizzled(e)/AllPairs(e) & \text{if } e = F(e_1, \ldots, e_k) \\ 1 & \text{if } e \text{ is an atom} \end{cases}$$

$$Swizzled(e) = \sum_{e_i \in \{e_1,\ldots,e_k\}} Swizzled(e_i) + |\{e_j | Overlap(e_i, e_j)\}|$$

$$Overlap(e, e') = Span(e) \cap Span(e') \neq \emptyset$$

$$Span(e) = [Min(UsedW(e)), Max(UsedW(e))]$$

$$AllPairs(e) = k \times (k-1) + \sum_{e_i \in \{e_1,\ldots,e_k\}} AllPairs(e_i)$$

Ranking Example. To illustrate how the ranking function identifies expressions that are likely matches for a user's intent, we look at two candidate expressions produced for "for all hours less than 20 sum the totalpay". One possible result expression is $e =$ `Sum(totalpay, True)` derived using a single pattern rule with a score value of 0.7. If all literal derivations have a weight of 1 then $ProdSc(e) = (0.85 \times 1)/1 = 0.85$. If the expression only uses last 3 words of the sentence we have $CoverSc(e) = \frac{1}{Max((9-3)^2,1)} = 0.027$. There is no interleaving; so the final score is 0.023.

Another possibility is $e' =$ `Sum(totalpay, Lt(hours, 20))` derived using a synthesis step over the sub-expressions $e_s =$ `Lt(hours, 20)` and $e'_s =$ `Sum(totalpay, □G2)`, each of which is derived using a pattern rule with score 0.7. Thus, $ProdSc(e') = (RScore(e'_s) \times SScore(e'_s))/2 + (RScore(e_s) \times SScore(e_s))/2$. In this case, $RScore(e'_s) = 0.7$ and $SScore(e'_s) = 0.49$. Thus, $ProdSc(e') = (0.7 \times 0.49)/2 + (0.85 \times 1)/2 = 0.597$. The result expression covers all relevant words ("for all" is not matched); thus we have $CoverSc(e') = 0.25$. The synthesis operation reordered words used in the expression but *did not* interleave them. Thus, the final score is 0.149 compared to 0.023 for the erroneous expression. So, we rank the desired expression first in the result list.

3.5 Full Algorithm Example

To illustrate how Algo. 1 is able to leverage the best characteristics of the rule based algorithm and the type based synthesis algorithm we consider the user description:

"for all hours less than 20 sum the totalpay"

Assuming the above description is issued over the sheet from Fig. 1 the algorithm will identify the words "hours" and "totalpay" as representing special column header symbols. As the dynamic programming algorithm progresses it will process "sum the totalpay" which, as a common way to express summation intents, will match rules of the form:

sum (all|the)* %C1 → Sum(□C1, True)
sum (all|the)* %C1 %2 → Sum(□C1, □G2)
sum (all|the)* %C1 → Sum(□C1, □G2)

When matching these rules Algo. 3 will fail on the second rule (as there are no matches for %2). However, the first and third rules will succeed, producing the expression `Sum(totalpay, True)` and the expression `Sum(totalpay, □G2)` with an unbound hole.

Algo. 1 will also process "hours less than 20" which matches the rule "%C1 less (than)* %L2" with %C1="hours" and %L2="20" to yield `Lt(hours, 20)`. When the algorithm reaches "hours less than 20 sum the totalpay", there are no rules that match the entire fragment; so the type based synthesis algorithm will be run with:

$$\{\text{Sum}(\text{totalpay}, \text{True}), \text{Lt}(\text{hours}, 20), \text{totalpay},$$
$$\text{Sum}(\text{totalpay}, \square\text{G2}), 20, \text{Lt}(\square\text{C1}, \square\text{G2}), \ldots\}$$

The type based synthesis algorithm will produce several expressions including the substitution of `Lt(hours, 20)` into the hole □G2 in the expression `Sum(totalpay, □G2)` to produce the the desired result expression:

Sum(totalpay, Lt(hours, 20))

The algorithm will continue to process increasingly larger sub-sequences of the input description, including the prefix "for all", until it reaches the complete input. At this

point the algorithm will produce the the union of all possible translations for the full input which includes the desired result. Finally, based on the ranking, the desired expression will be the top ranked result.

4. PROGRAMMING MODEL

Ambiguity Resolution. Since natural language is ambiguous and our translation algorithm may not be perfect, the output of our translation algorithm is a ranked set of likely programs in the DSL. Fig. 1 shows how multiple candidate results are generated and displayed to the user. In general we show up to three results that have confidence scores over a given threshold. However, to illustrate various features of the UI, the top three results are shown for the example regardless of their confidence score. We let the user select from among the synthesized programs in two orthogonal ways.

Each entry in the list has an annotated version of the user's description on the left (and the corresponding Excel function on the right). The annotated version of the description uses highlighting to show the words that were identified as column names or values from the sheet, red underlines to show misspelled words, and strike-through indicating words that were ignored when producing the corresponding expression. In Fig. 1 the first interpretation of the user description has identified *totalpay* as a column along with *baristas* and *capitol hill* as values that appear in the sheet contents. None of the parts of the description are struck out, indicating that all of the parts of the description were taken into account when producing the resulting expression. The next two entries in the list correspond to expressions that can be derived from the user description but are ranked as less likely because they ignore parts of the description, the statements about *baristas* and *capitol hill* respectively, shown by the strike-though on the words.

We transform each result expression into both Excel formulas and structured unambiguous English. Translation into Excel formulas is enabled by syntax-directed rewriting strategies that are standard in the compiler literature, and is done to avoid forcing users to learn our DSL. [1]

Translation into structured English is supported since many end users struggle with understanding Excel formulas. For this purpose, we associate pre-defined English descriptions with *both* the templates as well as the DSL operators. Thus, the same DSL expression can be paraphrased into different structured English descriptions depending on how the user originally specified the task. This allows our paraphrased English description to stay closer to the style of the user's original description. For the running example the paraphrased NL is "sum up the totalpay where title = barista and location = capitol hill". This description pops up in our UI when the user hovers over the displayed Excel formula.

Programming in Steps. Our interactive programming model for data analysis in spreadsheets allows the user to accomplish a sophisticated task using a sequence of steps. In each step, the user first selects the intended program from the ranked list of synthesized programs; the intended program is then executed and the state of the spreadsheet is changed as in a live programming model. The user may then proceed to the next step in the task pipeline. The sequence of programs produced can be automatically executed to update the output values if the user changes any input in the spreadsheet. This sequence of programs can also be executed on any similar spreadsheets.

Programs in our DSL change the spreadsheet state in one of two ways. One way is to generate a new scalar or vector value that is placed at the location of the current cursor as in Fig. 1. A program may lead to creation of a new value that can be used in subsequent programs to incrementally perform tasks. For instance after computing the totalpay sum for the capitol hill baristas the user may want to know what fraction this is of the overall payroll. One way to do this is to compute the total payroll in cell `I3` with the description "column H total" and then perform the division "divide I2 by I3". Alternatively the user can combine these steps into a single description "divide I2 by the total of column H".

Another way the spreadsheet state can change is by creating active selections or adding emphasis to the result of a query that selects certain rows/cells in the spreadsheet. The act of activating selections or adding emphasis (such as bold, color, etc.) to the spreadsheet values changes the meaning of implicit row/cell references in subsequent operations.

Creating emphasis can be likened to creating new view definitions (if the intended emphasis does not already exist in the spreadsheet) or updating view definitions (if the intended emphasis already exists). Generating emphasis might be the preferred mode of operation if the view definition created might be used in multiple subsequent operations. For example to compute totalpay sum for the chefs and the baristas the user could first create a new set of the chef totalpay values "color the chef totalpay red", followed by extending this set with the barista totalpay values "color the totalpay for the baristas red", and finally adding all the values in this set "add up all the values in the red cells".

On the other hand highlighting can be likened to creating an anonymous view definition, which can be referenced implicitly in a subsequent description. Highlighting might be a preferred mode of operation if the view definition (i.e., the query result) created is only used in the subsequent description. For example, a user could compute the totalpay sum for the capitol hill baristas by first selecting all the relevant rows "select the rows for the capitol hill baristas" and then adding the totalpay up for the selection, "get the totalpay from the selected rows and sum it".

Inter-operability with PBE. Our PBNL methodology uses the same principles used in the PBE methodology for spreadsheet data manipulation [9], namely: design of an appropriate DSL, translation algorithm for mapping specifications (whether examples or natural language) to likely DSL programs, and ranking those programs. This allows easy integration of PBE features like Excel's Flash Fill [1, 7] as one of the sequencing steps to achieve a sophisticated task in our programming model. For example, consider a table with three columns: Paper title, a string of comma-separated authors, and year of publication (as obtained from Google Scholar for some researcher R). Our DSL cannot express the task "How many papers have R as the first author". However, the user can start out by extracting the first authors in a new column by simply giving an example and invoking Flash Fill followed by using NLyze to complete the task.

[1]We target our DSL as an intermediate representation as Excel formulas do not naturally express a number of constructs, such as `Or(b1, b2)` which in Excel is often implemented using `IF(b1+b2, 1, 0)`. These limitations hinder both translation and paraphrasing.

5. IMPLEMENTATION AND EVALUATION

In this section we evaluate our approach for translating natural language to spreadsheet formulas. We implemented all of the algorithms described as a Excel add-in (using C#) called NLyze. The Excel product team provided us with 4 spreadsheets that contained data from conceptually different areas, employee payrolls, inventory management, country facts, and sales invoices. These sheets provided a variety for the vocabulary and implicit relations that users might have in a natural language description. Using questions asked in online forms and data from the Excel product team, we constructed 40 tasks involving conditional reduce/selection operations, lookup tasks, arithmetic formula, and combinations of these operations. We took before and after screen shots of performing these tasks on one or more of the 4 spreadsheets and, via an online crowd-sourcing, asked users to look at the before/after images and describe what they would tell a human to do in order to accomplish the illustrated task. This resulted in a suite of 3570 natural language descriptions over the different tasks and spreadsheets.

In addition to containing a range of tasks, our data contains many variations on how a person may express each task. We performed clustering on the natural language inputs for a given intent based on the orders of the column names/values and word similarity [41]. On average we found 37.7 distinct clusters for each intent, which demonstrates the wide range of ways different users express the same intent. The template algorithm needs to see representative examples covering all of these variations. However, the combined algorithm can leverage the synthesis sub-algorithm to successfully interpret these variations even when the training data, and thus translation rules, do not contain any example from some clusters. To construct the rules we performed a random 70/30 split of collected natural language descriptions and used the 70% split to build a set of 105 rules.

5.1 Overall Performance

We begin by evaluating the running time, precision, and recall of the overall translation algorithm. Tab. 2 shows the performance of the translation algorithm on the test data for the 4 different spreadsheets and, in the last row, cumulative results over all the sheets.

The second column in Tab. 2 shows the average time taken to translate from a user description to the Excel formula results. The translation is fast enough, between one and two hundredths of a second on average, to support a real-time search style UI where the user can see, in real time, the current results and how they change as the input changes.

The next two columns in Tab. 2 show how often the desired expression is (1) the top ranked result shown to the user and (2) how often the desired expression is in the top three results shown to the user. For each spreadsheet the desired result is top ranked for over 90% of the inputs and over all sheets/inputs the desired expression is the top ranked result in 94% of the inputs. In the search style UI a user can easily scan the first few results, with the help of expression paraphrasing provided by the system when needed, to select the desired formula. The last column in Tab. 2 shows the percentage of user descriptions for which the system was able to generate the correct result anywhere in the results list. There are a small number of cases (under 2%) where the correct formula can be produced but where it is not ranked in the top three results. The last row shows that the recall rate of the algorithm is 98% over all the sheets and inputs.

Sheet	Avg. Time	Top Rank	Top 3	All
Sheet #1	0.010s	94.4%	96.7%	97.5%
Sheet #2	0.015s	95.5%	97.5%	99.1%
Sheet #3	0.007s	94.5%	97.3%	97.9%
Sheet #4	0.019s	90.7%	96.7%	96.9%
All Sheets	0.011s	94.1%	97.1%	98.2%

Table 2: The average time per translation is shown in *Avg. Time*. The *Top Rank*, *Top 3*, and *All* columns show the percentage of task descriptions for which the intended program is respectively: the top ranked, in the top 3, or anywhere in the results.

A standard metric for evaluating the overall performance of a translation system is the *F1 score* (also called F-score or F-measure) which is the harmonic mean of the precision and recall values: $2\times(precision\times recall)/(precision+recall)$. This measure provides a balanced combination of both the recall (how often the system produces an answer) and the precision (how often a returned answer is correct) of the system. Using the recall and precision results we see that, in practice, the system provides the user with an Excel formula that matches their intent with a 97.6% F1-score success rate. The high value for the F1 score shows that the system performs well from a users perspective by both consistently producing an answer and a correct one.

5.2 End-User Evaluation

To further evaluate the ability of our system to successfully interpret user descriptions in practice we performed a second study where end-users were able to use the NLyze system. In this study we distributed the Excel add-in to a small group of users, provided a tutorial on the features supported by the system and demonstrated a number of sample tasks. We then asked them to use our add-in on a spreadsheet of their choice. We logged this usage and collected 62 task description/spreadsheet pairs.

The task descriptions seen in this study included vocabulary not in the training set (e.g. "nonzero othours") and contained more composition of expressions (e.g. combining lookup inside arithmetic operations) than the crowd-sourced data collected for training. Despite these challenges the translation algorithm (using the same set of rules that were derived in the first study) was able to generate the desired expression as the top candidate for 90.3% of the inputs, 93.5% for the top 3, and 95.1% anywhere in the result list. Further, the study participants reported that the ability to see multiple results, along with which words were used/ignored and paraphrasing of the expressions, gave them confidence that the Excel formula they selected would perform the desired calculation.

5.3 Evaluation of Algorithm Components

The overall system combines the basic pattern rule algorithm (Algo. 3) and the synthesis algorithm (Algo. 2) to improve recall, and leverages the ranking methodology (§3.4) to improve precision. To understand how these components contribute to the overall behavior, we examine results for each step in the process.

The first row of Tab. 3 shows the result of running just the baseline pattern rule algorithm (Algo. 3) and using only the scores of the rules and the production tree to rank the

result expressions. The recall rate for the algorithm is high, 89.8%, which indicates that the rule language and matching semantics (e.g., the *slack* tokens and kleene star matching of the `OptPat` patterns) allow for the construction of a rule set that covers most of the ways that users express their intents. The second row of Tab. 3 shows the result of running just the baseline synthesis algorithm (Algo. 2) and using only the scores of the rules to rank the result expressions. As expected the recall rate is higher than for the template algorithm, 98.2%, but the rates at which the desired result is the top ranked (67.4%) is substantially lower. However, for both algorithms there are many cases where the desired expression is not generated or where the desired expression is not the top ranked result (or is not in the top-3).

The third row in Table 3 shows the result of running the combined translation algorithm (Algo. 1) using only the scores of the rules and the production tree to rank the result expressions. The combination of the rule and the synthesis algorithms succeeds in producing the correct DSL expressions in cases where the baseline pattern algorithm alone fails. Thus, the recall (the *All* column) increases by 8.4% to near the limit of what is possible at 98.2% while the top ranked and top-3 scores are much higher than possible with the synthesis algorithm alone. However, the simplistic ranking based just on the production history is not enough to distinguish the desired expression from other expressions. Thus, the *Top 3* and *Top* ranked rates remain a respectable but still unsatisfactorily low 89.4% and 75.1% respectively.

The final row in Tab. 3 shows the results after the addition of the full ranking methodology from §3.4 and is equivalent to full algorithm (i.e., same as in Tab. 2). The results show that, as expected, the improved ranking does not affect the overall recall, unchanged at 98.2%, but it drastically improves the rate at which the desired expression is *Top 3* and the *Top* ranked to 97.1% and 94.1% respectively. As a result we conclude that all thee components of the combined interpretation, the pattern rule algorithm, the synthesis algorithm, and the ranking, are critical to the overall results. Further, the combination of these techniques result in a recall rate of 98.2%, the placement of the desired expression as the top ranked result for 94.1% of the inputs, and in the top 3 results for 97.1% of the inputs. From a user standpoint these results imply that the user will almost always find their desired result in the top 3 results and for roughly 19 out of 20 inputs the desired result will be the first suggestion.

Extensions	Top Rank	Top 3	All
Pattern Rule Only	74.0%	83.6%	89.8%
Synthesis Only	67.4%	85.6%	98.2%
Pattern Rule & Synthesis	75.1%	89.4%	98.2%
Complete Algorithm	94.1%	97.1%	98.2%

Table 3: The performance of the base pattern rule (semantic parsing) and synthesis (keyword programming) algorithms and the impact of the combination. The *Top Rank*, *Top 3*, and *All* columns show the percentage of user descriptions where the intended expression is in the category.

6. RELATED WORK

Programming by demonstration (PBD) based systems, which use a trace of a task performed by a user, and **programming by example** (PBE) systems [8, 9], which learn from a set of input-output examples, have been used to enable end-user programming for a variety of domains. For PBD these domains include text manipulation [19] and table transformations [13]. Recent work on PBE by Gulwani et.al. has included domains for manipulating strings [7, 35], numbers [36], and tables [10]. Both PBD and PBE based techniques struggle when the desired transformations involve conditional operations. This is because the number of examples required increases rapidly with the number of conditionals. In several scenarios, even a large number of examples fail to precisely characterize the desired conditionals. In contrast, natural language based approaches perform well for both simple and multi-conditional operations.

Yessenov et. al. [40] present a **programming by steps** system, where the user provides mock examples at each step. Our steps programming model allows use of both natural language and examples, but more significantly, its support for emphasis, highlighting, and implicit referencing allows easy communication between various steps.

There has been extensive research on developing **natural language interfaces to databases** (NLIDB) [2, 30]. NaLIX [21, 22] presents a natural language query interface to an XML database using the structure of the natural language parse tree derived from the user description. PRECISE [11, 33, 34] translates *semantically tractable* NL questions into corresponding SQL queries by matching tokens identified in the user description with the schema of the database to produce the SQL query. The tabular and frequently relational nature of spreadsheet data makes the task of translating natural language descriptions to spreadsheet formula somewhat similar. However, the spreadsheet domain requires different design choices than in the database domain because of (a) lack of explicit data schema, (b) the interactive and live programming nature of the environment, and (c) the need to support non-developer users (who, as we found out, use much less structured and colloquial English than what is present in previous data sets).

Keyword programming refers to the process of translating a set or sequence of keywords into a program. This program may consist either of operations in an existing programming language [31, 39] or a DSL constructed for a specific class of tasks [24, 25]. Keyword programming approaches generally have high recall but low precision and rely on the user to select the intended program from among multiple possible candidate result programs. Le et.al. present a system [20] that extracts keywords along with some dataflow relations from natural language descriptions and extends them to programs in the underlying DSL. Their system has high precision, but is specialized to the domain of smartphone automation scripts.

Semantic parsing [28] uses NLP based techniques to construct a program from natural language. Several approaches have been presented, namely: syntax directed [14], those that use parse trees [4], SVM driven [15], combinatory categorial grammars [18, 42, 43], and dependency-based semantics [23, 32] among others. These approaches typically have high precision but lower recall and are sensitive to grammatically incorrect/ill-formed descriptions.

We combine techniques from keyword programming and semantic parsing in a novel unified framework to achieve both high precision and high recall. Our translation algorithm applies the pattern rule based semantic parsing approach as much as possible (to achieve high precision) while interleaving it with type based synthesis (a keyword pro-

gramming approach) for those parts of the input that are outside the set of common/known patterns (to achieve high recall). Furthermore, any advances in semantic parsing (or keyword programming) can be easily plugged into our unified framework to further improve our results.

7. CONCLUSION AND FUTURE WORK

Program synthesis is the task of automatically synthesizing a program in some underlying *domain-specific language* from a given *specification* using some *search technique* [6]. The traditional view of program synthesis has been to synthesize programs from formal and complete specifications [27, 37]. Recent work has shown how to synthesize programs from examples, which are an ambiguous specification of the user's intent, using ranking and interactivity [7–9]. This line of work, which is targeted for end users, has been relatively more successful, and is a promising direction to enable end users to program computers. We build over this recent line of work to synthesize programs from natural language, which constitutes another useful form of (ambiguous) specification.

Our domain-specific language for spreadsheet data analysis/manipulation combines functional and compositional nature of SQL queries along with formatting based side effects that are common in spreadsheet environments. Our synthesis technique combines and builds over ideas from keyword programming (in PL/HCI communities) and semantic parsing (in NLP/DB communities). Our interaction model is end-user friendly with support for ambiguity resolution, sequencing of communicating DSL programs, and integration with programming by example techniques.

Opportunities for future work include the application of our generic translation algorithm to other data manipulation domains like text processing and table formatting. It would also be interesting to consider incorporating *similarity matching* techniques [3, 17] for column names and spreadsheet values and extending the lookup capability of NLyze to search over collections of *web tables* in addition to a user's own tables (as in [35]).

Acknowledgments

We thank the Excel product team, James McCaffrey, and Ben Zorn for providing valuable data and feedback.

References

[1] Flash Fill: Excel 2013 feature. http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html.

[2] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - An introduction. *CoRR*, 1995.

[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *VLDB*, 2009.

[4] R. Ge and R. J. Mooney. A statistical semantic parser that integrates syntax and semantics. In *CoNLL*, 2005.

[5] M. Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals*, 2009.

[6] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

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

[8] S. Gulwani. Synthesis from examples: Interaction models and algorithms. *SYNASC*, 2012. Invited Paper.

[9] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *CACM*, 2012.

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

[11] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a natural language interface to complex data. *TODS*, 1978.

[12] L. Jiang and Z. Su. Osprey: A practical type system for validating dimensional unit correctness of C programs. In *ICSE*, 2006.

[13] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[14] R. Kate, Y. W. Wong, and R. Mooney. Learning to transform natural to formal languages. In *AAAI*, 2005.

[15] R. J. Kate and R. J. Mooney. Using string-kernels for learning semantic parsers. In *ACL*, 2006.

[16] A. Ko, B. Myers, and H. Aung. Six learning barriers in end-user programming systems. In *VL/HCC*, 2004.

[17] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[18] T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. Lexical generalization in CCG grammar induction for semantic parsing. In *EMNLP*, 2011.

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

[20] V. Le, S. Gulwani, and Z. Su. Smartsynth: synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[21] Y. Li, H. Yang, and H.Jagadish. NaLIX: An interactive natural language interface for querying XML. In *SIGMOD*, 2005.

[22] Y. Li, H. Yang, and H.Jagadish. Constructing a generic natural language interface for an XML database. In *EDBT*, 2006.

[23] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL*, 2011.

[24] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *CHI*, 2007.

[25] G. Little and R. C. Miller. Translating keyword commands into executable code. In *UIST*, 2006.

[26] G. Little and R. C. Miller. Keyword programming in Java. In *ASE*, 2009.

[27] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 1980.

[28] R. J. Mooney. Learning for semantic parsing. In *CICLing*, 2007.

[29] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 1980.

[30] N. Nihalani, S. Silakari, and M. Motwani. Natural language interfce for database: A brief review. *IJCSI*, 2011.

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

[32] H. Poon. Grounded unsupervised semantic parsing. In *ACL*, 2013.

[33] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[34] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, 2003.

[35] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *VLDB*, 2012.

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

[37] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.

[38] WordNet. http://wordnet.princeton.edu.

[39] D. M. L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *POPL*, 2005.

[40] K. Yessenov, S. Tulsiani, A. K. Menon, R. C. Miller, S. Gulwani, B. W. Lampson, and A. Kalai. A colorful approach to text processing by example. In *UIST*, 2013.

[41] W. Yih and V. Qazvinian. Measuring word relatedness using heterogeneous vector space models. In *NAACL HLT*, 2012.

[42] L. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, 2005.

[43] L. S. Zettlemoyer and M. Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL*, 2007.