FlashExtract: A Framework for Data Extraction by Examples

Vu Le [*]
University of California at Davis
vmle@ucdavis.edu

Sumit Gulwani
Microsoft Research Redmond
sumitg@microsoft.com

Abstract

Various document types that combine model and view (e.g., text files, webpages, spreadsheets) make it easy to organize (possibly hierarchical) data, but make it difficult to extract raw data for any further manipulation or querying. We present a general framework FlashExtract to extract relevant data from semi-structured documents using examples. It includes: (a) an interaction model that allows end-users to give examples to extract various fields and to relate them in a hierarchical organization using structure and sequence constructs. (b) an inductive synthesis algorithm to synthesize the intended program from few examples in *any* underlying domain-specific language for data extraction that has been built using our specified algebra of few core operators (map, filter, merge, and pair). We describe instantiation of our framework to three different domains: text files, webpages, and spreadsheets. On our benchmark comprising 75 documents, FlashExtract is able to extract intended data using an average of 2.36 examples in 0.84 seconds per field.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Languages, Algorithms, Human Factors

Keywords Program Synthesis, End-user Programming, Programming by Examples

1. Introduction

The IT revolution over the past few decades has resulted in two significant advances: the digitization of massive amounts of data and widespread access to computational devices. However, there is a wide gap between access to rich digital information and the ability to manipulate and analyze it.

Information is available in documents of various types such as text/log files, spreadsheets, and webpages. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data model. However, this makes it extremely hard to extract the underlying data for several tasks such as data processing, querying, altering the presentation view, or transforming data to another storage format. This has led to development of various domain-specific technologies for data extraction. Scripting languages like Perl, Awk, Python have been designed to support string processing in text files. Spreadsheet systems like Microsoft Excel allow users to write macros using a rich built-in library of string and numerical functions, or to write arbitrary scripts in Visual Basic/.NET programming languages. Web technologies like Xquery, HTQL, XSLT can be used to extract data from webpages, but this has the additional burden of knowing the underlying document structure.

Existing programmatic solutions to data extraction have three key limitations. First, the solutions are domain-specific and require knowledge/expertise in different technologies for different document types. Second, they require understanding of the entire underlying document structure including the data fields that the end user is not interested in extracting and their organization (some of which may not even be visible in the presentation layer as in case of webpages). Third, and most significantly, they require knowledge of programming. The first two aspects create challenges for even programmers, while the third aspect puts these solutions out of reach of the vast majority of business end users who lack programming skills. As a result, users are either unable to leverage access to rich data or have to resort to manual copy-paste, which is both time-consuming and error prone.

In this paper, we address the problem of developing a uniform end-user friendly interface to support data extraction from semi-structured documents of various types. Our methodology includes two key novel aspects: a uniform user interaction model across different document types, and a generic inductive program synthesis framework.

Uniform and End-user Friendly Interaction Model Our extraction interface supports data extraction via examples. The user initiates the process by providing a nested hierarchical definition of the data that he/she wants to extract using standard structure and sequence constructs. The user then provides examples of the various data fields and their relationships with each other. An interesting aspect is that this model is independent of the underlying document type. This is based on our observation that different document types share one thing in common: a two-dimensional presentation layer. We allow users to provide examples by highlighting two-dimensional regions on these documents. These regions indicate either the fields that the user wants to extract or structure/record boundaries around related fields.

Inductive Program Synthesis Framework To enable data extraction from examples, we leverage inductive program synthesizers that can synthesize scripts from examples in an underlying domain-specific language (DSL). The key technical contribution of this paper is an inductive program synthesis framework that allows easy development of inductive synthesizers from mere definition of DSLs (for data extraction from various document types). We describe an expressive algebra containing four core operators: map, filter, merge, and pair. For each of these operators, we define its sound and complete generic inductive synthesis algorithms parameterized by the

[*] Work done during two internships at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PLDI '14, June 09 - 11 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594333

```
DLZ - Summary Report
"Sample ID:,""5007-01"""
"Sample Date/Time:,""Wednesday, May 30, 2006 00:43:51"""
Intensities
"I/S,""Analyte"",""Mass"",""Conc. Mean"",""Unit"",""Conc. SD"",""RSD"",""Mean"""
"|-,""Be"",9,0.070073,""ug/L"",0.009,12.542,121.334"
"|>,""Sc"",45,,""ug/L"",,,404615.043"
"| ,""Ti"",48,10.653153,""ug/L"",0.847,7.949,181379.200"
"|-,""Se"",82,1.009204,""ug/L"",0.026,2.613,457.487"
"|-,""Sr"",88,20.163079,""ug/L"",2.005,9.943,718014.023"
"|>,""Rh"",103,,""ug/L"",,,438976.176"
DLZ - Summary Report
"Sample ID:,""5007-02"""
"Sample Date/Time:,""Wednesday, May 30, 2006 01:02:38"""
Intensities
"I/S,""Analyte"",""Mass"",""Conc. Mean"",""Unit"",""Conc. SD"",""RSD"",""Mean"""
"| ,""Mn"",55,71.705740,""ug/L"",0.350,0.489,2428667.736"
"| ,""Co"",59,0.131132,""ug/L"",0.004,3.315,3606.816"
"|-,""Ba"",138,129.339264,""ug/L"",3.088,2.387,4648771.382"
"|-,""Hf"",178,,""ug/L"",,,338359.496"
"| ,""Tl"",205,2.876992,""ug/L"",0.730,25.380,129217.588"
"| ,""Pb"",208,3.671043,""ug/L"",0.026,0.702,228830.402"
```

Figure 1: Extracting data from a text file using FlashExtract.

```
Show: 20    1-20 Next >
Title / Author                                                          Cited by  Year
Finding bugs with a constraint solver
D Jackson, M Vaziri                                                      197      2000
ACM SIGSOFT Software Engineering Notes 25 (5), 14-25
Associating synchronization constraints with data in an object-oriented language
M Vaziri, F Tip, J Dolby                                                 176      2006
ACM SIGPLAN Notices 41 (1), 334-345
Some Shortcomings of OCL, the Object Constraint Language of UML.
M Vaziri, D Jackson                                                      81       2000
TOOLS (34), 555-562
Model checking software systems: A case study
JM Wing, M Vaziri-Farahani                                               68       1995
ACM SIGSOFT Software Engineering Notes 20 (4), 128-139
```

Figure 2: Extracting data from a Google Scholar webpage.

operator's arguments. As a result, the synthesis designer simply needs to define a DSL with two features: (a) It should be expressive enough to provide appropriate abstractions for data extraction for the underlying document type, (b) It should be built out of the operators provided by our core algebra. The synthesis algorithm is provided for free by our framework. This is a significant advance in the area of programming by examples, wherein current literature [11, 12] is limited to domain-specific synthesizers.

This paper makes the following contributions:

- We present a uniform and end-user friendly interaction model for data extraction from examples (§3). This eliminates the need to learn domain-specific scripting technologies for various document types. It also eliminates the need to understand the document's internal data model and its representation.
- We present a rich algebra of operators for data extraction DSLs and a modular inductive synthesis strategy (which is sound, complete, and practically efficient) for each of these operators (§4). This allows development of inductive synthesizers for data extraction from various document types from a mere definition of an appropriate DSL using these operators. This eliminates the need to develop specialized program synthesis algorithms.
- We present three useful instantiations of our framework to the domains of text files, webpages, and spreadsheets (§5). Each of these improves the state of the art for data extraction from respective document types.
- We present detailed experimental evaluation that illustrates the effectiveness of the three instantiations of our general framework on 25 documents each (§6). Each of these instantiations were able to synthesize the desired extraction script using an average of 2.36 examples in 0.84 seconds per field.

We start out with motivating examples for data extraction from various document types and illustrate our user interaction model.

2. Motivating Examples

In this section, we motivate some data extraction tasks across different document types and illustrate how FlashExtract can be used to automate the various tasks from examples.

Text Extraction

EXAMPLE 1. *Consider the text file in Fig. 1 (taken from a help forum thread*[1]*) that contains a sequence of sample readings, where each sample reading lists various "analytes" and their characteristics (analyte intensities). The user wants to extract the highlighted fields into an Excel spreadsheet in order to perform some analysis. Accomplishing this task by creating a one-off Perl script appears daunting, especially for a non-programmer.*

Suppose the user only wants to extract the *analyte names* (magenta regions starting with instance "Be") and their *mass* (violet regions starting with instance "9"). The user starts with the analyte names. She highlights the first two regions "Be" and "Sc" as examples in magenta color. FlashExtract synthesizes an extraction program and uses it to highlight all other analyte instances. The user inspects and approves the highlighted result because it matches the intended sequence. She then moves to the mass field and repeats the process with violet color. FlashExtract can now automatically relate the respective instances from the magenta sequence and the violet sequence, and can generate a two-column Excel table if desired.

Now suppose the user also wants to extract the *conc. mean* (blue regions including instance "0.070073"). After one example, FlashExtract mistakenly includes the string ",""ug/L,,,404615.043" to the result (this should be null). To exclude this region, the user draws a red line through it to mark it as a *negative example*, and FlashExtract refines the learning with the new information. It then produces the correct result and the user stops providing any further examples. Although the third sequence contains fewer regions, FlashExtract is still able to relate it to the other two automatically because it is the only sequence containing null regions.

In case FlashExtract cannot relate different field regions, or does so incorrectly, the user can intervene by marking a structure boundary around related regions. For instance, the user may highlight the first yellow region as the structure boundary for the intensity of the first analyte. FlashExtract is able to infer similar yellow regions that group other intensities. If the user wants to further organize the analyte intensities into different samples, she creates the outer green regions. The user can then add the sample ID (orange field, such as "5007-01") to these green structures.

Once the highlighting process has been completed, the user can obtain the data (in different formats such as XML file or Excel table) and its associated data extraction program. The user may run the program on other similar files to extract data in the same output format without giving any additional examples.

Note that the user can extract data in any field order (we only demonstrated one such order). For example, the green regions can be highlighted before the yellow regions, which in turn can be highlighted before the violet regions. The top-down order is generally recommended and has higher chance of success (because an inner field knows who is its parent). Furthermore, the highlighting does not necessarily need to follow the actual file structure; it just needs to be consistent. For instance, the user may want the green structures to begin at ""Sample ID", or the yellow structures to end in the middle of the lines, say before "ug/L".

We can combine FlashExtract with existing end-user programming technologies to create new user experiences. For instance, integration of FlashExtract with FlashFill [10] allows users to both extract and transform the highlighted fields using examples, and possibly push the changes back to the original document. As an example, after highlighting using FlashExtract, the user can easily

[1] http://www.excelforum.com/excel-programming/608284-read-txt-file.html

	Amount	Investigator
Advanced Materials Processing	Analysis Center (AMPAC)	
	$24,500.00	Coffey, Kevin R.
Subtotal	$24,500.00	
Biomolecular Science Center		
	$155,458.00	Khaled, Annette R.
	$71,777.00	Zervos, Antonis S.
Subtotal	$227,235.00	
Education (ED)		
	$272,349.00	Daire, Andrew
	$57,500.00	Little, Mary E.
	$272,349.00	Young, Mark E.
Subtotal	$602,198.00	

Figure 3: Extracting data from a semi-structured spreadsheet.

change the precision of *conc. mean* (blue field) or the casing of *analytes* (magenta field) using FlashFill.

Webpage Extraction

EXAMPLE 2. *Google Scholar website* (http://scholar.google.com) *has author pages containing list of all publications. A publication consists of its title, list of authors, venue, number of citations, and year of publication. Fig. 2 shows an excerpt from a page of a researcher Mandana Vaziri.*

Suppose the user wants to find all publication titles in which Dr. Vaziri is the first author. She can use FlashExtract to extract the publication title (blue) and just the first author (magenta) fields into an Excel spreadsheet, where she can utilize the native spreadsheet functionality to sort by first author field. A key design principle behind FlashExtract is to provide a uniform interaction model independent of the underlying document type. The user interacts with webpages exactly as with text files. That is, the user gives examples for the desired fields by using colored highlighting over the rendered webpage. She does not need to understand the underlying structure of the webpages (as is required in the case of querying languages such as XPath, XQuery).

Now suppose the user wants to extract publication titles along with the list of all authors. Extracting list of all authors is technically challenging because the comma separated list of all authors is represented as a string in a single `div` tag. However, FlashExtract can be used to highlight each author individually. Its underlying DSL is expressive enough to allow extracting list of regions in a text field of a single HTML node.

FlashExtract can be used to group a publication and all its authors together. The user may make this relation explicit by highlighting green regions. The same applies to the yellow regions that group author sequences. Once FlashExtract has produced the program, the user may run it on other scholar pages to perform the same task for other authors.

Spreadsheet Extraction

EXAMPLE 3. *Consider the semi-structured spreadsheet* `"Funded - February#A835C.xlsx"` *from the EUSES benchmark [15] shown in Fig. 3. Suppose the user wants to (a) add up the values in the Amount column (excluding the values in the subtotal rows), and (b) plot values in the Amount column grouped by department name.*

The user highlights few examples of the amount field (magenta), department field (yellow), and the record boundaries (green). FlashExtract is then able to highlight all other similar instances and creates a new relational table view. For task (a), the user can now simply add up all the values in the amount column by using the native spreadsheet SUM function over the new relational view (instead of having to write a complicated conditional arithmetic macro over the original semi-structured spreadsheet view). Task (b) is easily accomplished using the popular "Recommended Charts" feature in Excel 2013, wherein Excel automatically suggests relevant charts over user's data. Note that this feature only works when the data is properly formatted as a single relational table—it does not work on the original semi-structured spreadsheet.

If the user later decides to also extract the investigator name (blue), she can simply provide an example. As before, once all interactions are recorded, the output view can be automatically updated if the user continues to edit and maintain the original ad-hoc format consistently.

$$
\begin{aligned}
\text{Schema } M &= S \mid T \\
\text{Structure } T &= \texttt{Struct}\ (\mathit{identifier}: E_1, \ldots, \mathit{identifier}: E_n) \\
\text{Element } E &= f \mid S \\
\text{Sequence } S &= \texttt{Seq}(f) \\
\text{Field } f &= [\mathit{color}]\ \tau \mid [\mathit{color}]\ T
\end{aligned}
$$

Figure 4: The language of schema for extracted data.

3. User Interaction Model

Our user interaction model for data extraction requires the user to provide an *output data schema* and highlight examples of *regions* (in the document) that contain the desired information. The final result is a *schema extraction program* that extracts the desired data from the document as an instance of the output schema. We next define these aspects in more detail.

Output Schema

The final product of an extraction task is a nested organization of the extracted data using standard structure and sequence constructs. Fig. 4 defines the language for the output schema. The output schema is either a sequence S over some field f, or a structure T with named elements $E_1, \ldots, E_n$. Each element E corresponds to either a field f or to a sequence S. Each field f is either an atomic type τ (also referred to as a *leaf field*) or a structure T. Each field f is associated with a unique color (denoted $f.\texttt{Color}$).

For example, the schemas for the two extraction tasks discussed in Ex. 1 are presented below. The first one represents a sequence of yellow structures, each of which contains a magenta Analyte field and a violet Mass field. The second one represents organization of many more fields referenced in the task illustrated in Fig. 1.

(1) Seq([yellow] Struct(Analyte : [magenta] Float,
    Mass : [violet] Int))

(2) Seq([green] Struct(SampleID : [orange] String,
    Intensities : Seq([yellow] Struct(
        Analyte : [magenta] String,
        Mass : [violet] Int,
        CMean : [blue] Float))))

Note that the schema language does not allow a sequence to be directly nested inside another sequence. It requires a colored structure construct in between them. The structure serves as the boundary for the learning of the inner sequence.

DEFINITION 1 (Ancestor). *We say that a field f_1 is an* ancestor *of field f_2 if f_2 is nested inside f_1. For notational convenience, we say that $\perp$, which stands for the top-level data schema definition, is an ancestor of every field. Additionally, f_1 is a* sequence-ancestor *of f_2 if there is at least one sequence construct in the nesting between f_1 and f_2. Otherwise, f_1 is a* structure-ancestor *of f_2. We say that $\perp$ is an ancestor of every other field.*

In the second schema above, the yellow structure is the structure-ancestor of leaf fields Analyte, Mass, and CMean. The top-level green structure is the sequence-ancestor of the yellow structure.

function Run *(schema extraction program Q, schema M, document D) : schema instance* **is**

```
1   CR := ∅
2   foreach field f in M in top-down topological order do
3       R̃ := Run(Q(f), D, CR)
4       CR := CR ∪ {(f.Color, R) | R ∈ R̃}
5   if CR is inconsistent with M then return ⊥
6   else return Fill(M, D.Region)
```

function Run *(extraction program (f', P) of field f, document D, highlighting CR): f-regions* **is**

```
7   R̃' := (f' = ⊥)? {D.Region} : CR[f'.Color]
8   return ⋃_{R' ∈ R̃'} [[P]]R'          /* execute P on R' */
```

Algorithm 1: Execution semantics of extraction programs.

We categorize ancestral relationship in order to invoke appropriate synthesis algorithms.

Regions

DEFINITION 2 (Region). *A* region R *of a document is some two-dimensional portion over the visualization layer of that document that the user is allowed to highlight in some color. We use the notation* f-region *to denote any region that the user highlights in* f.`Color`*. Any region R that is associated with a leaf field (also referred to as a* leaf region*) has some value associated with it, which is denoted by* R.`Val`*. For a document D, we use the notation* D.`Region` *to denote the largest region possible in D.*

In case of text files, any region is represented by a pair of two character positions within the file and consists of all characters in between (these positions may or may not be within the same line). The value of such a region is the string of all characters in between those positions.

In case of webpages, a leaf region is represented by either an HTML node (the value of such a region is the text value associated with that node) or a pair of character positions within the text content of an HTML node (the value of such a region is the string of all characters in between those positions). A non-leaf region is represented by an HTML node.

In case of spreadsheets, a leaf region is represented by a single cell (and its value is the cell's content), while a non-leaf region is represented by a pair of cells (and consists of the rectangular region determined by those cells).

DEFINITION 3 (Highlighting). *A* highlighting `CR` *of a document D is a collection of colored regions in D. It can also be viewed as a function that maps a color to all regions of that color in D. We say that a highlighting* `CR` *is* consistent *with a data scheme M if the following conditions hold.*

- *For any two regions (in* `CR`*), either they don't overlap or one is nested inside the other.*
- *For any two fields f_1 and f_2 in M such that f_1 is an ancestor of f_2, each f_2-region R_2 is nested inside some f_1-region R_1.*
- *For any two fields f_1 and f_2 in M such that f_1 is a struct-ancestor of f_2, there is at most one f_2-region inside a f_1-region.*
- *For every leaf field f in M, the value of any f-region in* `CR` *is of type f.*

Schema Extraction Program

FlashExtract synthesizes extraction programs for individual fields and combines them into a schema extraction program following the structure imposed by the output schema. FlashExtract also leverages the schema's structure to simplify the learning of individual fields. In particular, it relates a field f to one of its ancestors, whose extraction program (in case of a non-$\bot$ ancestor) defines learning boundaries for f (*i.e.*, each f-region must reside inside one of these boundaries).

$$\texttt{Fill}\big(\texttt{Struct}\ (id_1\ E_1, \ldots, id_n\ E_n), R\big) = \textbf{new}\ \texttt{Struct}\big(\{id_1 = \texttt{Fill}(E_1, R), \ldots, id_n = \texttt{Fill}(E_n, R)\}\big)$$

$$\texttt{Fill}\big(\texttt{Seq}(f), R\big) = \textbf{new}\ \texttt{Seq}\big(\texttt{Map}(\lambda R' : \texttt{Fill}(f, R'), \texttt{Subregions}(R, \texttt{CR}[f.\texttt{Color}])\big)$$

$$\texttt{Fill}([color]\ \texttt{Val}, R) = \texttt{Subregion}(R, \texttt{CR}[color]).\texttt{Val}$$

$$\texttt{Fill}([color]\ T, R) = \texttt{Fill}(T, \texttt{Subregion}(R, \texttt{CR}[color]))$$

$$\texttt{Fill}(_, \bot) = \bot$$

Figure 5: Semantics of `Fill`.

DEFINITION 4 (Extraction Programs). *A* schema extraction program *Q for a given schema M is represented as a map from each field f in M to a field extraction program, denoted $Q(f)$.*

A field extraction program *of field f is a pair (f', P), where f' (possibly $\bot$) is some ancestor field of f and P is either a* SeqRegion program *that extracts a sequence of f-regions from inside a given f'-region (in case f' is a sequence-ancestor of f), or is a* Region program *that extracts a single f-region from inside a given f'-region (in case f' is a struct-ancestor of f).*

The execution semantics of a schema extraction program is defined in Algorithm 1. FlashExtract executes the field extraction program corresponding to each field f in a top-down order and updates the document highlighting `CR` using the returned list of f-regions $\tilde{R}$ (lines 2–4). For each field f, it first finds the set of regions $\tilde{R}'$ determined by the ancestor f' (line 7), and then computes all f-regions by executing the field extraction program on each region R' in $\tilde{R}'$ (line 8). Once `CR` has been fully constructed, it generates a schema instance from the nesting relationship defined in the output schema M, using the `Fill` function (line 6).

Fig. 5 defines the semantics of `Fill` recursively. Each definition takes a schema construct and a region corresponding to one of its ancestor fields, and returns a construct instance by recursively applying `Fill` functions on its descendants. `CR`$[c]$ returns all regions whose color is c. `Subregions`$(R, \tilde{R})$ returns the ordered set of regions from $\tilde{R}$ that are nested inside R. `Subregion`$(R, \tilde{R})$ returns the region from $\tilde{R}$ that is nested inside R; if no such region exists, $\bot$ is returned. Note that if `CR` is consistent with M, there is at most one such region. We assume the presence of an API for checking the nestedness of two regions.

Example-based User Interaction

Having defined all necessary concepts, we are now ready to discuss the way a user interacts with FlashExtract to extract their desired data. The user first supplies the output data schema. Then, for each field f in the schema (in an order determined by the user), the user simply provides sufficient number of examples of field instances of field f by highlighting appropriate regions in the document using f.`Color`. Our user interface supports standard mouse click, drag, and release gestures.

When the user provides examples for a field f, FlashExtract synthesizes a field extraction program for field f (using Algorithm 2) that is *consistent* with the provided examples, and executes it to identify and highlight other regions in f.`Color`. (See Def. 5 for a formal notion of consistency.) If the user is happy with the inferred highlighting, she can commit the results (the field f is said to have been *materialized* at that point of time), and then proceed to another (non-materialized) field. Otherwise, the user may provide any additional examples.

We say that a field f has been *simplified* if there exists a materialized field f' such that f' is a structure-ancestor of f. The examples for a non-simplified field consist of positive instances and optionally negative instances of regions that lie completely within the regions of the immediate ancestor field that has been

```
function SynthesizeFieldExtractionProg (Document D,
Schema M, Highlighting CR, Field f, Regions R̃1, Regions R̃2) is
    /* R̃1, R̃2 denote positive, negative instances */
1   foreach ancestor field f' of f in schema M do
2       if f' isn't materialized ∧ f' ≠ ⊥ then continue
3       R̃ := (f' = ⊥)? {D.Region} : CR[f'.Color]
4       if f' is a sequence-ancestor of f then
5           ex := ∅
6           foreach R ∈ R̃ s.t. Subregions(R, R̃1 ∪ R̃2) ≠ ∅
            do
7               ex := ex ∪
                {(R, Subregions(R, R̃1), Subregions(R, R̃2))}
8           P̃ := SynthesizeSeqRegionProg(ex)
9       else /* f' is a structure-ancestor of f */
10          ex := ∅
11          foreach R ∈ R̃ s.t. Subregion(R, R̃1) ≠ ⊥ do
12              ex := ex ∪ {(R, Subregion(R, R̃))}
            P̃ := SynthesizeRegionProg(ex)
        foreach P ∈ P̃ do
            CR' := CR ∪ {(f.Color, R) | R ∈ ⟦P⟧R', R' ∈ R̃}
13          if CR' is consistent with M then return (f', P)
14  return ⊥
```

Algorithm 2: Synthesize a field extraction program.

materialized. If the user is not happy with the inferred highlighting, the user provides additional positive instances (of regions that FlashExtract failed to highlight) or negative instances (of unintended regions that FlashExtract highlighted) and the synthesis process is repeated. The examples for a simplified field consist of at most a single positive instance (possibly null) inside each region of the immediate ancestor field that has been materialized. If the user is not happy with the inferred highlighting, the user provides additional examples and the synthesis process is repeated.

The example-based interaction is enabled by the procedure SynthesizeFieldExtractionProg described in Algorithm 2, which takes as input a document D, a schema M, a highlighting CR of the document that is consistent with M, a non-materialized field f, a set of positive instances $\tilde{R}_1$, and a set of negative instances $\tilde{R}_2$ (which is empty in case field f has been simplified). The procedure SynthesizeFieldExtractionProg returns a program P such that (a) P is consistent with the examples and (b) the updated highlighting that results from executing P is consistent with the schema M. Line 2 finds a suitable ancestor f' from CR that forms the learning boundary for f. The loops at lines 6 and 11 group the input examples into boundaries imposed by f'-regions. Depending on the relationship between f and f', FlashExtract invokes an appropriate API provided by our inductive synthesis framework. In particular, it invokes SynthesizeSeqRegionProg (line 8) to learn a program for extracting a sequence of f-regions in an f'-region (if f' is a sequence-ancestor of f), or SynthesizeRegionProg (line 12) to learn a program for extracting a single f-region in an f'-region (if f' is a structure-ancestor of f). Both SynthesizeSeqRegionProg and SynthesizeRegionProg actually return a sequence of programs of the right type. The loop at line 12 selects the first program P in this sequence (if it exists) that ensures that the updated highlighting that results from executing P is consistent with the schema M. We describe this framework and its APIs in the next section.

An interesting aspect of the above-mentioned interaction is the order in which the user iterates over various fields. FlashExtract is flexible enough to let users extract various fields in any iteration order. This is especially useful when the user dynamically decides to update the data extraction schema (e.g., extract more fields). Iterating over fields in a bottom-up ordering offers an interesting advantage. It allows FlashExtract to guess the organization of leaf field instances by looking at their relative order (thereby obviating the need to provide examples for any non-leaf field.) While this is successful in most cases, it may not be able to deal with cases where field instances may be null. On the other hand, iterating over fields in a top-down topological order requires the user to provide examples for each field (including non-leaf fields), but it offers three advantages: (a) it provides an easy visualization for the user to inspect the results of the organization of various non-leaf field instances, (b) it provides greater chance of success since the synthesis task for a field can now be enabled relative to any ancestor field as opposed to the entire document, (c) it may also entail having to provide fewer examples for any field that is nested inside another field whose instances have all been identified. Hence, if the leaf field instances are never null and the user does not need help with identifying representative examples, the user may simply provide few examples for each leaf field and FlashExtract may be able to automatically infer the organization of the various leaf field instances. Otherwise, we recommend that the user iterates over fields in a top-down topological order.

4. Inductive Synthesis Framework

In this section, we describe a general framework for developing the inductive synthesis APIs (namely, SynthesizeSeqRegionProg and SynthesizeRegionProg) that enable the example-based user interaction model discussed in the previous section. We build this framework over the inductive synthesis methodology proposed by Gulwani et.al. [12] of designing appropriate DSLs and developing algorithms to synthesize programs in those DSLs from examples. However, we go one step further. We identify an algebra of core operators that can be used to build various data extraction DSLs for various document types (§4.2). We also present modular synthesis algorithms for each of these operators in terms of the synthesis algorithms for its (non-atomic) arguments (§4.3)—this enables automatic generation of synthesis algorithms for any DSL that is constructed using our algebra of core operators. We start out by formalizing the notion of a data extraction DSL.

4.1 Data Extraction DSLs

A data extraction DSL is represented by a tuple (G, N_1, N_2). G is a grammar that defines data extraction strategies. It contains definitions for various non-terminals N. Each non-terminal N is defined as a ranked collection of rules (also referred to as N.RHSs) of the same type. The type of a non-terminal is the type of its rules. A rule consists of a fixed expression or an operator applied to other non-terminals of appropriate types or fixed expressions. The type of a rule is the return type of the fixed expression or the operator that constitutes the rule.

We say a non-terminal is *synthesizable* if each of its rules either (a) involves an operator from our core algebra applied to fixed expressions or synthesizable non-terminals, or (b) involves an operator that is equipped with an inductive synthesis algorithm of its own (*i.e.*, domain-specific operators), or (c) fixed expressions. N_1 is a distinguished (top-level) synthesizable non-terminal of type sequence of regions. N_2 is another distinguished (top-level) synthesizable non-terminal of type region.

An expression generated by a non-terminal of type T can be viewed as a program with return type T. Note that the expressions generated by N_1 represent SeqRegion programs and expressions generated by N_2 represent Region programs. The DSL expressions may involve one distinguished free variable R_0 (of type Region) that denotes the input to the top-level SeqRegion or Region programs. Any other free variable that occurs in a DSL expression must be bound to some lambda definition that occurs in a higher level expression.

A state σ of a program P is an assignment to all free variables in P. We use the notation $\{x \leftarrow v\}$ to create a state that maps variable x to value v. We use the notation $\sigma[v/x]$ to denote setting the value of variable x to value v in an existing state σ. We use the notation $[\![P]\!]\sigma$ to denote the result of executing the program P in state σ.

Next, we discuss the core operators in our algebra that can be used to build data extraction DSLs.

4.2 Core Algebra for Constructing Data Extraction DSLs

Our core algebra is based around certain forms of map, filter, merge, and pair operators. The pair operator (which returns a scalar) constitutes a *scalar expression*, while the other operators (which return a sequence) constitute a *sequence expression*.

Decomposable Map Operator A `Map` operator has two arguments $\lambda x : F$ and S, where S is a sequence expression of type $\texttt{List}\langle T\rangle$ and F is some expression of type T' and uses an additional free variable x. The return type of `Map` is $\texttt{List}\langle T'\rangle$. $\texttt{Map}(\lambda x : F, S)$ has the standard semantics, wherein it applies function F to each element of the sequence produced by S to construct the resultant sequence.

$$[\![\texttt{Map}(\lambda x : F, S)]\!]\sigma \;=\; [t_0, \ldots, t_n], \text{ where}$$
$$n = |Y' - 1|, t_i = [\![F]\!](\sigma[Y'[i]/x]), Y' = [\![S]\!]\sigma$$

We say that a `Map` operator is *decomposable* (*w.r.t.* the underlying DSL, which defines the language for F and S) if it has the following property: For any input state σ and a sequence Y, there exists a sequence Z such that

$$\forall F, S : Y \sqsubseteq [\![\texttt{Map}(F, S)]\!]\sigma \implies Z \sqsubseteq [\![S]\!]\sigma \wedge [\![\texttt{Map}(F, Z)]\!]\sigma = Y$$

where $\sqsubseteq$ denotes the subsequence relationship. Let `Decompose` be a function that computes such a witness Z, given σ and Y. It has the following signature:

$$\texttt{Map.Decompose} : (\mathit{Region} \times \texttt{List}\langle T'\rangle) \rightarrow \texttt{List}\langle T\rangle$$

The `Decompose` function facilitates the reduction of examples for `Map` operator to examples for its arguments F and S, thereby reducing the task of learning the desired `Map` expression from examples to the sub-tasks of learning F and S expressions from respective examples.

Filter Operators Our algebra allows two kinds of filter operators over sequences, one that selects elements based on their properties (`FilterBool`), and the other one that selects elements based on their indexes (`FilterInt`).

A `FilterBool` operator has two arguments $\lambda x : B$ and S, where S is a sequence expression of type $\texttt{List}\langle T\rangle$ and B is a Boolean expression and uses an additional free variable x. The return type of `FilterBool` is $\texttt{List}\langle T\rangle$. $\texttt{FilterBool}(\lambda x : F, S)$ has the standard filtering semantics: it selects those elements from S that satisfy B. For example, if S is the set of all lines in Ex. 1, then the expression $\texttt{FilterBool}(\lambda x : \texttt{EndsWith}([\text{Number, Quote}], x), S)$ selects all yellow lines. The predicate $\texttt{EndsWith}([\text{Number, Quote}], x)$ returns true iff the string x ends with a number followed by a double quote.

A `FilterInt` operator has three arguments: a non-negative integer `init`, a positive integer `iter`, and a sequence expression S. Its return value also has the same type as that of S. The `FilterInt` operator takes every `iter` elements from S starting from `init` as the first element. Its semantics is as follows:

$$[\![\texttt{FilterInt}(\texttt{init}, \texttt{iter}, S)]\!]\sigma \;=\; \textbf{let } L = [\![S]\!]\sigma \textbf{ in}$$
$$\texttt{Filter}\big(\lambda x : (\texttt{indexof}(L, x) - \texttt{init})\%\texttt{iter} = 0, L\big)$$

For example, $\texttt{FilterInt}(1, 2, S)$ selects all elements at odd indices from a sequence.

The two kinds of filter operators can be composed to enable sophisticated filtering operations.

Merge Operator A `Merge` operator takes as input a set of n sequence expressions, each of which is generated by the same non-terminal A (of some type of the form $\texttt{List}\langle T\rangle$). The return value of `Merge` also has the same type as that of A. The `Merge` operator combines the results of these n expressions together—this is useful when a single expression cannot extract all intended regions. This operator is a disjunctive abstraction and allows extraction of multiple-format field instances by merging several single-format field instances. Its semantics is as follows:

$$[\![\texttt{Merge}(A_1, \ldots, A_n)]\!]\sigma \;=\; \texttt{MergeSeq}([\![A_1]\!]\sigma, \ldots, [\![A_n]\!]\sigma)$$

The `MergeSeq` operation merges its argument sequences with respect to the order of their elements' locations in the original file.

Pair Operator A `Pair` operator has two arguments A and B and has the following standard pair operator semantics.

$$[\![\texttt{Pair}(A, B)]\!]\sigma \;=\; ([\![A]\!]\sigma, [\![B]\!]\sigma)$$

The pair operator allows constructing region representations from smaller elements. For example, we can create a text region from a pair of its start and end positions.

4.3 Modular Synthesis Algorithms

The API `SynthesizeSeqRegionProg` takes as input a set of examples, each of which consists of a triple $(R, \tilde{R_1}, \tilde{R_2})$, where R denotes the input region, $\tilde{R_1}$ denotes positive instances of the regions that should be highlighted within R, and $\tilde{R_2}$ denotes negative instances of the regions that should not be highlighted within R. The API `SynthesizeRegionProg` takes as input a set of examples, each of which consists of a pair (R, R'), where R denotes the input region and R' denotes the output region. Both these APIs return an ordered set of programs in the DSL, each of which is consistent with the provided examples. Fig. 6 contains the pseudo-code for these APIs, which we explain below.

The method `SynthesizeSeqRegionProg` first learns from only positive instances by invoking the learn method of the top-level sequence non-terminal N_1 (line 2) and then selects those programs that additionally satisfy the negative instances constraint (loop at line 4). The operator :: appends an element to a list. The method `SynthesizeRegionProg` simply invokes the learn method of the top-level region non-terminal N_2 (line 10).

The learn method for any non-terminal N simply involves invoking the learn method associated with its various rules (line 13) and then returning the ordered union of the sequences of the programs synthesized from each. The operator $+\!\!+$ performs list concatenation.

We next briefly describe the key insights behind the learn methods for the operators that constitute the various rules. The higher level key idea is to define them in terms of the learn methods of their arguments. This allows for a free synthesis algorithm for any data extraction DSL.

Learning Decomposible Map Operator The key idea here is to first find the witness Z_j for each example (σ_j, Y_j) using the operator's `Decompose` method (line 16). This allows us to split the problem of learning a map expression into two independent simpler sub-problems, namely learning of a scalar expression F(line 18), and learning of a sequence expression S (line 20) from appropriate examples. The returned result is an appropriate cross-product style composition of the results returned from the sub-problems (line 25).

Learning Merge Operator The key idea here is to consider all (minimal) partitioning of the examples such that the learn method of the argument for each partition returns a non-empty result. The set T (at line 28) holds all such minimal partitions. For each such partition (loop at line 30), we invoke the learn method of the argument for each class in the partition (line 31), and then appropriately combine

```
function SynthesizeSeqRegionProg (
Set⟨(Region, List⟨Region⟩, List⟨Region⟩)⟩ Q) : List⟨Prog⟩ is
1    Q' := {({R_0 ← R}, R̃_1) | (R, R̃_1, R̃_2) ∈ Q}
     /* learn with positive examples */
2    P̃ := N_1.Learn(Q')    /* start symbol for sequence */
3    P̃' := []
4    foreach P ∈ P̃ do
5        if (∃(R, R̃_1, R̃_2) ∈ Q : (⟦P⟧{R_0 ← R}) ∩ R̃_2 ≠ ∅) then
6            continue   /* P violate negative instances */
7        P̃' := P̃' :: P
8    return P̃'

function SynthesizeRegionProg (Set⟨(Region, Region)⟩ Q) :
List⟨Prog⟩ is
9    Q' := {({R_0 ← R}, R') | (R, R') ∈ Q}
10   return N_2.Learn(Q')    /* start symbol for region */

function N.Learn(Set⟨(State, T)⟩ Q) : List⟨Prog⟩ is
11   P̃ := []
12   foreach C ∈ N.RHSs do
13       P̃ := P̃ ++ C.Learn(Q)
14   return P̃

function Map.Learn (Set⟨(State, List⟨T⟩)⟩ Q) : List⟨Prog⟩ is
     /* Let F and S be the function and sequence
        arguments of Map. */
     Let Q be {(σ_j, Y_j)}_{1≤j≤m}
15   for j := 1..m do                   /* find witnesses Z */
16       Z_j := Map.Decompose(σ_j, Y_j)
17   Q_1 := {(σ_j[Z_j[i]/x], Y_j[i]) | 0 ≤ i < len(Z_j), 1 ≤ j ≤ m}
18   P̃_1 := F.Learn(Q_1)      /* learn Map's function F */
19   Q_2 := {(σ_j, Z_j) | 1 ≤ j ≤ m}
20   P̃_2 := S.Learn(Q_2)      /* learn Map's sequence S */
21   P̃ := []
22   foreach P_1 ∈ P̃_1 do
23       foreach P_2 ∈ P̃_2 do
24           P̃ := P̃ :: "Map(P_1, P_2)"
25   return CleanUp(P̃, Q)

function Merge.Learn (Set⟨(State, List⟨T⟩)⟩Q) : List⟨Prog⟩ is
     /* Let A be the non-terminal that forms the
        arguments of Merge. */
     Let Q be {(σ_j, Y_j)}_{1≤j≤m}
     /* X holds all possible subsets of examples */
26   X := {Q' | Q' = {(σ_j, Y'_j)}_{1≤j≤m},
               ∀1 ≤ j ≤ m : Y'_j ⊑ Y_j, A.Learn(Q') ≠ []}
27   Y := ⋃_{(σ_j,Y_j)∈Q} Y_j              /* all positive examples */
     /* T includes partitions that cover all examples */
28   T := {X' | X' is a minimal subset of X s.t.
               {Y'_j | (σ_j, Y'_j) ∈ Q', Q' ∈ X'} = Y}
29   P̃ := []
30   foreach X' ∈ T ordered by size do
         Let Q'_1, ..., Q'_n be the various elements of X'
31       P̃_1, ..., P̃_n := A.Learn(Q'_1), ..., A.Learn(Q'_n)
32       P̃ := P̃ ++ {"Merge(P_1, ..., P_n)" | ∀1 ≤ i ≤ n : P_i ∈ P̃_i}
33   return CleanUp(P̃, Q)

function FilterBool.Learn (Set⟨(State, List⟨T⟩)⟩ Q) :
List⟨Prog⟩ is
     /* Let B and S be the predicate and sequence
        arguments of FilterBool. */
35   P̃_1 := S.Learn(Q)                  /* learn sequence S */
36   Q' := {(σ[Y[i]/x], True) | (σ, Y) ∈ Q, 0 ≤ i < len(Y)}
37   P̃_2 := B.Learn(Q')                  /* learn filter B */
38   P̃ := []
39   foreach P_1 ∈ P̃_1 do
40       foreach P_2 ∈ P̃_2 do
41           P̃ := P̃ :: "FilterBool(P_1, P_2)"
42   return CleanUp(P̃, Q)

function FilterInt.Learn (Set⟨(State, List⟨T⟩)⟩ Q) :
List⟨Prog⟩ is
     /* Let S be the sequence argument of FilterInt. */
     Let Q be {(σ_j, Y_j)}_{1≤j≤m}
43   P̃_1 := S.Learn(Q)                  /* learn sequence S */
44   P̃ := []
45   foreach P_1 ∈ P̃_1 do
46       init := ∞
47       iter := 0
48       for j := 1 ... m do
49           Z_j := ⟦P_1⟧σ_j
50           init := Min(init, indexof(Z_j, Y_j[0]))
51           for i := 0 ... |Y_j| − 2 do
52               t := indexof(Z_j, Y_j[i+1]) − indexof(Z_j, Y_j[i])
53               if iter = 0 then iter := t
54               else iter := GCD(iter, t)
55       if iter = 0 then iter := 1
56       P̃ := P̃ :: "FilterInt(init, iter, P_1)"
57   return CleanUp(P̃, Q)

function Pair.Learn (Set⟨(State, (T_1, T_2))⟩ Q) : List⟨Prog⟩ is
     /* Let A and B be the two arguments of Pair. */
     Let Q be {(σ_j, (u_j, u'_j))}_{1≤j≤m}
58   Q_1 := {(σ_j, u_j)}_{1≤j≤m};  Q_2 := {(σ_j, u'_j)}_{1≤j≤m}
59   P̃_1 := A.Learn(Q_1)
60   P̃_2 := B.Learn(Q_2)
61   if P̃_1 = ∅ or P̃_2 = ∅ then return []
62   P̃ := []
63   foreach P_1 ∈ P̃_1 do
64       foreach P_2 ∈ P̃_2 do
65           P̃ := P̃ :: "Pair(P_1, P_2)"
66   return P̃

function CleanUp(List⟨Prog⟩ P̃, Set⟨(State, List⟨T⟩)⟩ Q) :
List⟨Prog⟩ is
67   P̃' := []
68   foreach i = 1 to |P̃| do
69       P := P[i]
70       incl := true
71       foreach k = 1 to |P̃| do
72           if (P̃[k] subsumes P w.r.t. Q) and ((P does not subsume
             P̃[k] w.r.t. Q) or k < i) then incl := false
73       if (incl = true) then P̃' := P̃' :: P
74   return P̃'
```

Figure 6: Modular inductive synthesis algorithm in FlashExtract.

the results. Although this algorithm is exponential in the size of the input set, it works efficiently in practice because the number of examples required for learning is very small in practice.

Learning Filter Operators The key idea in the learn method for `FilterBool` is to independently learn an ordered set $\tilde{P}_1$ of sequence expressions (line 35) and an ordered set $\tilde{P}_2$ of Boolean expressions (line 37) that are consistent with the given examples. The returned result is an appropriate cross-product style composition of the sub-results $\tilde{P}_1$ and $\tilde{P}_2$.

The key idea in the learn method for `FilterInt` is to first learn an ordered set $\tilde{P}$ of sequence expressions (line 43) that are consistent with the given examples. Then, for each such program, we learn the most strict filtering logic that filters as few elements as possible while staying consistent with the examples. In particular, we select `init` to be the minimum offset (across all examples) of the first element in Y_j in the sequence Z_j returned by executing the sequence program in the example state σ_j (line 50). We select `iter` to be the GCD of all distances between the indices of any two contiguous elements of Y_j in Z_j (line 54).

Learning Pair Operator The key idea here is to invoke the learn method of the first (second) argument at line 59 (line 60) to learn programs that can compute the first (second) element in the various output pairs in the examples from the respective inputs. The final result is produced by taking a cross-product of the two sets of programs that are learned independently (loop at line 63).

CleanUp Optimization An important performance and ranking optimization employed by the learn methods of various operators is use of the `CleanUp` method, which removes those programs that extract more regions than some retained program. More precisely, this method removes each of those (lower-ranked) programs from an ordered set of programs that is *subsumed* by some unremoved program (See Def. 6). Note that this does not affect the completeness property associated with the various learning methods (Th. 3). Furthermore, it implements an important ranking criterion that assigns higher likelihood to the scenario wherein the user provides consecutive examples from the beginning of any immediately enclosing ancestral region (as opposed to providing arbitrary examples).

4.4 Correctness

We now describe the correctness properties associated with our two key synthesis APIs: `SynthesizeSeqRegionProg` and `SynthesizeRegionProg`. First, we state some useful definitions.

DEFINITION 5. *(Consistency) A scalar program P (i.e., a program that returns a scalar value) is said to be* consistent *with a set $Q = \{(\sigma_j, u_j)\}_j$ of scalar examples if $\forall j : u_j = [\![P]\!]\sigma_j$. A sequence program P (i.e., a program that returns a sequence) is said to be* consistent *with a set $Q = \{(\sigma_j, Y_j)\}_j$ of sequence examples with positive instances if $\forall j : Y_j \subseteq [\![P]\!]\sigma_j$. A sequence program P is said to be* consistent *with a set $Q = \{(\sigma_j, Y_j, Y_j')\}_j$ of sequence examples with positive and negative instances if $\forall j : (Y_j \subseteq [\![P]\!]\sigma_j \wedge Y_j' \cap [\![P]\!]\sigma_j = \emptyset)$.*

DEFINITION 6. *(Subsumption) Given a set $Q = \{(\sigma_j, Y_j)\}_j$ of sequence examples with positive instances, and two sequence programs P_1, P_2 that are consistent with Q, we say that P_1* subsumes *P_2 w.r.t. Q if $\forall j : [\![P_1]\!]\sigma_j \subseteq [\![P_2]\!]\sigma_j$.*

The following two theorems hold.

THEOREM 1 (Soundness). *The programs P returned by `SynthesizeSeqRegionProg` and `SynthesizeRegionProg` are consistent with the input set of examples.*

The proof of Theorem 1 follows easily by induction (on the structure of the DSL) from similar soundness property of the learn methods associated with the non-terminals and core algebra operators.

THEOREM 2 (Completeness). *If there exists some program that is consistent with the input set of examples, `SynthesizeSeqRegionProg` (and `SynthesizeRegionProg`) produce one such program.*

The proof of Theorem 2 follows from two key observations: (a) The learn methods associated with the scalar non-terminals and the (scalar) Pair operator satisfy a similar completeness property. (b) The learn methods associated with the sequence non-terminals and the sequence operators of the core algebra satisfy a stronger completeness theorem stated below (Theorem 3).

THEOREM 3 (Strong Completeness). *The learn methods associated with the sequence non-terminals and the sequence operators of the core algebra (namely Map, FilterBool, FilterInt, and Merge) satisfy the following property: "For every sequence program P that is consistent with the input set of examples, there exists a program P' in the learned set of programs that subsumes P."*

The proof of Theorem 3 follows from induction on the DSL's structure. Note that the `CleanUp` optimization only removes those (lower-ranked) programs that are subsumed by other programs.

5. Instantiations

We now present instantiations of our general framework to three different data extraction domains: text files, webpages, and spreadsheets. For each domain, we define the notion of a region, the domain's underlying data extraction DSL, and discuss the implementation of its domain-specific learn methods.

5.1 Text Instantiation

A region in this domain is a pair of character positions in the input text file.

Language $\mathcal{L}_{\text{text}}$ Fig. 7 shows the syntax of $\mathcal{L}_{\text{text}}$, our data extraction DSL for this domain. The core algebra operators are in **bold**. We name the various `Map` operators differently in order to associate different `Decompose` methods with them. The non-terminal N_1 is a `Merge` operator over constituent sequence expressions SS. The non-terminal N_2 is defined as a `Pair` operator over two position expressions.

The position expression $\texttt{Pos}(x, p)$ evaluates to a position in string x that is determined by the attribute p (inspired by a similar concept introduced in [10]). The attribute p is either an absolute position k, or is the kth element of the position sequence identified by the regex pair rr which consists of two regexes r_1 and r_2. The selection order is from left-to-right if k is positive, or right-to-left if k is negative. The position sequence identified by (r_1, r_2) in string x, also referred to as $\texttt{PosSeq}(x, rr)$, is the set of all positions k in x such that (some suffix of the substring on) the left side of k matches with r_1 and (some prefix of the substring on) the right side of k matches with r_2. A regex r is simply a concatenation of (at most 3) tokens. A token T is a pre-defined standard character class such as alphabets, digits, colon character, etc. (We used 30 such tokens in our instantiation). We also define some context-sensitive tokens dynamically based on frequently occurring string literals in the neighborhood of examples highlighted by the user. For instance, in Ex. 1, our dynamically learned tokens include the string "DLZ - Summary Report" (which is useful for learning the green outer structure boundary) and the string ""Sample ID:,""" (which is useful to extract the orange sample ID).

The first rule of SS consists of a `Map` operator `Lines`**Map** that maps each line of a line sequence LS to a pair of positions within that line. The `Decompose` method for `Lines`**Map** takes as input a region R and a sequence of position pairs and returns the sequence of lines from R that contain the corresponding position pairs.

The second (third) rule of SS pairs each position x in a position sequence PS with a position that occurs somewhere on its right (left)

Disjuctive Pos Pair Seq $\mathbf{N_1} ::= \textbf{Merge}(SS_1, \ldots, SS_n)$

Pos Pair Region $\mathbf{N_2} ::= \textbf{Pair}(\texttt{Pos}(R_0, p_1), \texttt{Pos}(R_0, p_2))$

Pair Seq $SS ::= \texttt{Lines}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(x, p_1), \texttt{Pos}(x, p_2)), LS)$

$\mid \texttt{StartSeq}\textbf{Map}(\lambda x : \textbf{Pair}(x, \texttt{Pos}(R_0[x:], p)), PS)$

$\mid \texttt{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(R_0[:x], p), x), PS)$

Line Seq $LS ::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, BLS)$

Bool Line Seq $BLS ::= \textbf{FilterBool}(b, \texttt{split}(R_0, \text{'\textbackslash n'}))$

Position Seq $PS ::= \texttt{Lines}\textbf{Map}(\lambda x : \texttt{Pos}(x, p), LS)$

$\mid \textbf{FilterInt}(\texttt{init}, \texttt{iter}, \texttt{PosSeq}(R_0, rr))$

Predicate $b ::= \lambda x : \texttt{True}$

$\mid \lambda x : \texttt{\{Starts,Ends\}With}(r, x) \mid \lambda x : \texttt{Contains}(r, k, x)$

$\mid \lambda x : \texttt{Pred\{Starts,Ends\}With}(r, x) \mid \lambda x : \texttt{PredContains}(r, k, x)$

$\mid \lambda x : \texttt{Succ\{Starts,Ends\}With}(r, x) \mid \lambda x : \texttt{SuccContains}(r, k, x)$

Position Attribute $p ::= \texttt{AbsPos}(k) \mid \texttt{RegPos}(rr, k)$

Regex Pair $rr ::= (r_1, r_2)$ Regex $r ::= T\{0, 3\}$

Token $T ::= C+ \mid \texttt{DynamicToken}$

Figure 7: The syntax of $\mathcal{L}_{\text{text}}$, the DSL for extracting text files.

side. The notation $R_0[x:]$ ($R_0[:x]$) denotes the suffix (prefix) of the text value represented by R_0 starting (ending) at position x. The `Decompose` method associated with `StartSeq`**Map** (`EndSeq`**Map**) takes as input a region R and a sequence of positions and maps each position k in the input sequence to the string $R[k:]$ ($R[:k]$).

The line sequence non-terminal LS uses a nested combination of `FilterInt` and `FilterBool`. The various choices for predicate b (used in `FilterBool`) have the expected semantics. For example, $\texttt{StartsWith}(r, x)$ asserts if line x starts with regex r, while $\texttt{Contains}(r, k, x)$ asserts if line x contains k occurrences of regex r. We also take hints from preceding and succeeding lines via `Pred*` and `Succ*` predicates. For example, $\texttt{PredStartsWith}(r, x)$ asserts if the line preceding x in the input text file ends with regex r.

The position sequence non-terminal PS includes expressions that select a position within each line of a line sequence (using the `Lines`**Map** operator) or that filter positions returned by the `PosSeq` operator (using the **FilterInt** operator).

EXAMPLE 4. *Below is a program in $\mathcal{L}_{\text{text}}$ for extracting the yellow regions in Ex. 1 (from the top-level region of the entire file).*

$\textit{Lines}\textbf{Map}(\lambda x : \textbf{Pair}(\textit{Pos}(x, p_1), \textit{Pos}(x, p_2)), LS)$, *where*
$p_1 = \textit{AbsPos}(0), p_2 = \textit{AbsPos}(-1)$,
$LS = \textbf{FilterInt}(0, 1,$
$\textbf{FilterBool}(\lambda x : \texttt{EndsWith}(\textit{[Number, Quote]}, x), \texttt{split}(R_0, \text{'\textbackslash n'})))$

The `FilterBool` operator takes all the lines in the document and selects only those that end with a number and a quote. The `FilterInt` operator does not do any filtering (`init` = 0, `iter` = 1); it simply passes the result of `FilterBool` to LS. The map function in `Lines`**Map** returns the entire input line (`AbsPos` (0) denotes the beginning of the line, while `AbsPos` (-1) denotes the end of the line). The `Lines`**Map** operator thus returns a sequence identical to LS, which is the yellow sequence.

EXAMPLE 5. *Below is a program for extracting the magenta regions in Ex. 1 (from the top-level region of the entire file).*

$\textit{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\textit{Pos}(R_0[:x], p), x), PS)$, *where*
$p = \textit{RegPos}\big(([\texttt{DynamicTok}(,\text{""})], \epsilon), -1\big)$
$PS = \textbf{FilterInt}(0, 1, \textit{PosSeq}(R_0, (r_1, r_2))$, *and*
$r_1 = [\texttt{DynamicTok}(,\text{""}), \textit{Word}]$,
$r_2 = [\texttt{DynamicTok}(\text{""},), \textit{Number}, \textit{Comma}]$,

FlashExtract recognizes the prefixes (,"") and suffixes ("",) of the given examples as frequently occurring substrings and promotes them to dynamic tokens. The `PosSeq` operator returns the sequence of all end positions of the magenta sequence (since each of these have an r_1 match on the left and an r_2 match on the right). Note that there are other positions that either have an r_1 match on the left (such as the position before the number in "Sample ID:;""5007-01"""), or have an r_2 match on the right (such as the position after the character L in ""ug/L"",0.0009), but not both; hence, these positions are not selected by the `PosSeq` operator. Since `FilterInt` does not filter any elements, PS is the same sequence returned by the regex pair. The map function in `EndSeq`**Map** takes each end position in PS and finds its corresponding start position specified by p, which is the first position from the right (k = -1) that matches the dynamic token (,"") on the left side. The result is the magenta sequence.

Disjuctive Seq $\mathbf{N_1} ::= \textbf{Merge}(NS_1, \ldots, NS_n)$

$\mid \textbf{Merge}(SS_1, \ldots, SS_n)$

Region $\mathbf{N_2} ::= \texttt{XPath} \mid \textbf{Pair}(\texttt{Pos}(R_0, p_1), \texttt{Pos}(R_0, p_2))$

Node Seq $NS ::= \texttt{XPaths}$

Pos Pair Seq $SS ::=$

$\texttt{SeqPair}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(x.\texttt{Val}, p_1), \texttt{Pos}(x.\texttt{Val}, p_2)), ES)$

$\mid \texttt{StartSeq}\textbf{Map}(\lambda x : \textbf{Pair}(x, \texttt{Pos}(R_0[x:], p)), PS)$

$\mid \texttt{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(R_0[:x], p), x), PS)$

Element Seq $ES ::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, \texttt{XPaths})$

Position Seq $PS ::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, \texttt{PosSeq}(R_0, rr))$

Figure 8: The syntax of $\mathcal{L}_{\text{web}}$, the DSL for extracting webpages. Definitions of p and rr are similar to those in Fig. 7.

EXAMPLE 6. *If the magenta field is wrapped within the yellow structure, one of its extraction programs is as follows:*

$\textbf{Pair}(\textit{Pos}(R_0, p_1), \textit{Pos}(R_0, p_2))$, *where*
$p_1 = \langle[\texttt{DynamicTok}(,\text{""})], \epsilon, 1\rangle, \quad p_2 = \langle\epsilon, [\texttt{DynamicTok}(\text{""},)], 1\rangle$

Since the yellow field is the structure-ancestor of the magenta field, FlashExtract learns a `Pair` operator to extract a magenta region within a yellow region. The start position of this pair is the first position from the left (k = 1) that matches (,"") on the left side (r_1), and the end position is the first position from the left that matches ("",) on the right side (r_2). This program is simpler than the one in Ex. 5, because it exploits the separation determined by the program for the yellow field.

Domain-Specific Learn Methods The learning of Boolean expression b is performed using brute-force search. The learning of position attribute expressions p is performed using the technique described in prior work [10].

5.2 Webpage Instantiation

A region in this domain is either an HTML node, or a pair of character positions within the text property of an HTML node.

Language $\mathcal{L}_{\text{web}}$ Fig. 8 shows the syntax of the DSL $\mathcal{L}_{\text{web}}$ for extracting data from webpages. `XPath` (`XPaths`) denote an XPath expression that returns a single HTML node (a sequence of HTML nodes) within the input HTML node. Position attribute p and regex pair rr are similar to those in the text instantiation DSL $\mathcal{L}_{\text{text}}$. Our token set additionally includes dynamic tokens that we create for the various HTML tags seen in the input webpage.

The non-terminal N_1 represents expressions that compute a sequence of HTML nodes or a sequence of position pairs within HTML nodes. The non-terminal N_2 represents expressions that compute a HTML node or a position pair within a HTML node. The design of $\mathcal{L}_{\text{web}}$ is inspired by the design of $\mathcal{L}_{\text{text}}$. HTML nodes in $\mathcal{L}_{\text{web}}$ play a similar role to that of lines in $\mathcal{L}_{\text{text}}$. We use XPath expressions to identify relevant HTML elements instead of using regular expressions to identify appropriate lines.

$$
\begin{aligned}
\text{Disjuctive Cell Pair Seq } \mathbf{N_1} &::= \textbf{Merge}(PS_1, \ldots, PS_n) \\
&\mid \textbf{Merge}(CS_1, \ldots, CS_n) \\
\text{Cell Pair Region } \mathbf{N_2} &::= \textbf{Pair}(\texttt{Cell}(R_0, c_1), \texttt{Cell}(R_0, c_2)) \\
&\mid \texttt{Cell}(R_0, c) \\
\text{Pair Seq } PS &::= \texttt{StartSeq}\textbf{Map}(\lambda x : \textbf{Pair}(x, \texttt{Cell}(R_0[x:], c)), CS) \\
&\mid \texttt{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Cell}(R_0[:x], c), x), CS) \\
\text{Cell Sequence } CS &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, CE) \\
&\mid \texttt{CellRow}\textbf{Map}(\lambda x : \texttt{Cell}(x, c), RS) \\
\text{Row Sequence } RS &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, RE) \\
\text{Cell Attribute } c &::= \texttt{AbsCell}(k) \mid \texttt{RegCell}(cb, k) \\
\text{Cell Split Seq } CE &::= \textbf{FilterBool}(cb, \texttt{splitcells}(R_0)) \\
\text{Row Split Seq } RE &::= \textbf{FilterBool}(rb, \texttt{splitrows}(R_0)) \\
\text{Cell Boolean } cb &::= \lambda x : \texttt{True} \mid \lambda x : \texttt{Surround}(T\{9\}, x) \\
\text{Row Boolean } rb &::= \lambda x : \texttt{True} \mid \lambda x : \texttt{Sequence}(T+, x)
\end{aligned}
$$

Figure 9: The syntax of $\mathcal{L}_{\text{sps}}$, the DSL for extracting spreadsheets.

The non-terminal SS represents expressions that generate a sequence of position pairs by mapping each HTML node in a sequence ES to position pairs (SeqPair**Map** operator) or by pairing up each position in a sequence PS of positions with another position computed relative to it (StartSeq**Map** and EndSeq**Map** operators).

Domain-specific Learn Methods We need to define learn methods for XPath and XPaths from example HTML nodes. This is a well-defined problem in the data mining community, called wrapper induction (see §7). We implemented a learn method that generalizes example nodes to path expressions by replacing inconsistent tags at any depth with "*", and additionally incorporates common properties of example nodes. These properties include the number of children, their types, the number of attributes and their types. The result is a list of XPath expressions, ranging from the most specific to the most general.

5.3 Spreadsheet Instantiation

A region in this domain is a rectangular region represented by a pair of cells or a single cell.

Language $\mathcal{L}_{\text{sps}}$ Fig. 9 shows the syntax of our DSL. The non-terminal N_1 represents expressions that extract a sequence of cell pairs or a sequence of cells from a given spreadsheet. The non-terminal N_2 represents expressions that extract a cell pair or a single cell from a given spreadsheet.

$\mathcal{L}_{\text{sps}}$ is inspired by $\mathcal{L}_{\text{text}}$ and $\mathcal{L}_{\text{web}}$. The notion of a row in $\mathcal{L}_{\text{sps}}$ is similar to that of a line in $\mathcal{L}_{\text{text}}$ and an HTML node in $\mathcal{L}_{\text{web}}$. Just as Boolean expressions in $\mathcal{L}_{\text{text}}$ help filter lines by matching their content against regular expressions, the row Boolean expression rb in $\mathcal{L}_{\text{sps}}$ selects spreadsheet rows by matching contents of consecutive cells inside a row against some token sequence. In addition, the cell Boolean expression cb selects cells by matching contents of the cell and its 8 neighboring cells against some 9 tokens.

As their name suggests, the two functions splitcells and splitrows split a spreadsheet region into a sequence of cells (obtained by scanning the region from top to down and left to right) and into a sequence of rows respectively, on which cell and row Boolean expressions can be applied.

Domain-specific Learn Methods The learning of Boolean expression cb and rb is performed using brute-force search. The learning of cell attribute expressions c is performed in a manner similar to that of position attribute expressions p in $\mathcal{L}_{\text{text}}$.

We need to define learn methods for domain-specific top-level non-terminals c, cb and rb. To learn c, we simply perform brute-force search to find all matching cell expression cb. Learning the row expression rb is similar.

6. Evaluation

We implemented FlashExtract framework and its three instantiations (described in §5) in C#. We conducted all experiments on a machine running Windows 7 with Intel Core i7 2.67GHz, 6GB RAM. Via this evaluation, we seek to answer the following questions related to effectiveness of FlashExtract.

- Can FlashExtract describe DSLs that are expressive enough for extraction tasks on real world files?
- How many examples are required to extract the desired data?
- How efficient is FlashExtract in learning extraction programs?

Real-world Benchmarks We collected 75 test documents in total, 25 for each of the three domains. The text file domain is very broad. A text file might be relatively structured such as a log file, or loosely structured as in text copied and pasted from webpages or PDF documents. We selected a representative benchmark set that includes a few files for each kind. Additionally, we also included some benchmarks from the book "Pro Perl Parsing" [9], which teaches how to use Perl to parse data.

For the webpage domain, we used the benchmark from [20], which describes SXPath, an extension of XPath to perform queries on Web documents. This benchmark includes 25 e-commerce popular websites with different underlying structures. For each of the website, they have two test cases corresponding to the HTML elements of the product name and the product price. In addition to these, we add a test case for the region covering all product information, and another test case for the actual price number (ignoring other texts that may occur in the price element such as "sale", "$" or "USD").

For the spreadsheet domain, we obtained 25 documents from two sources: benchmark used in previous work on spreadsheet transformation [13] (we selected those 7 documents from this benchmark that were associated with non-trivial extraction tasks), and EUSES corpus [15].

Experimental Setup For each document, we wrote down an appropriate schema describing the type of the hierarchical data inside the document, and we manually annotated all instances for the various fields in that schema to precisely define the extraction task. We used FlashExtract to learn the extraction programs for each field. Recall that we can learn the extraction logic for a field f by relating it to any of its ancestors. Among these, relating to $\bot$, is typically the hardest (in contrast, relating to one of the other ancestors can exploit the separation that has already been achieved by the extraction logic for that ancestor).

We wrote a script to simulate user interaction with FlashExtract to measure its effectiveness in the above-mentioned hardest scenario. Let $\tilde{R}$ denote all manually annotated instances for a field f. The simulator starts out with an empty set of negative instances and a singleton set of positive instances that includes the first region in $\tilde{R}$, and repeats the following process in a loop. In each iteration, the simulator invokes FlashExtract to synthesize a field extraction program from the various examples. If FlashExtract fails to synthesize a program, the simulator declares failure. Otherwise, the simulator executes the program to find any mismatch *w.r.t.* the golden result $\tilde{R}$. If no mismatch exists, the simulator declares success. Otherwise, the simulator adds the first mismatched region as a new example: it is added as a positive instance if the mismatched region occurs in $\tilde{R}$ but is not highlighted in the execution result of the previous interaction; otherwise the mismatch is added as a negative example. Furthermore, the simulator also adds all new highlighted regions that occur before the new example as positive instances.

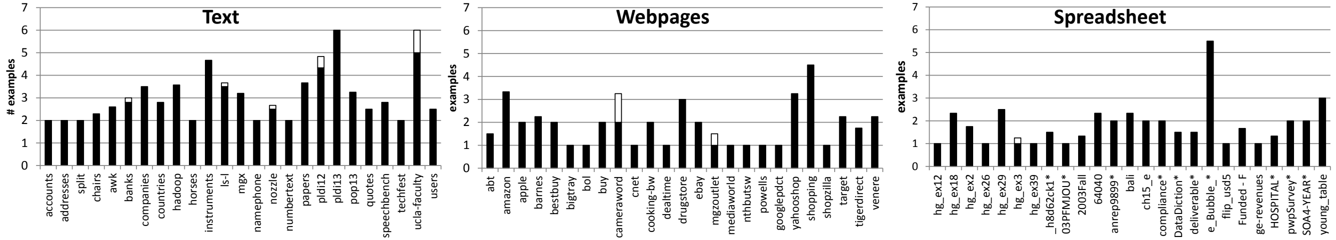

Figure 10: Average number of examples (solid/white bars represent positive/negative instances) across various fields for each document.

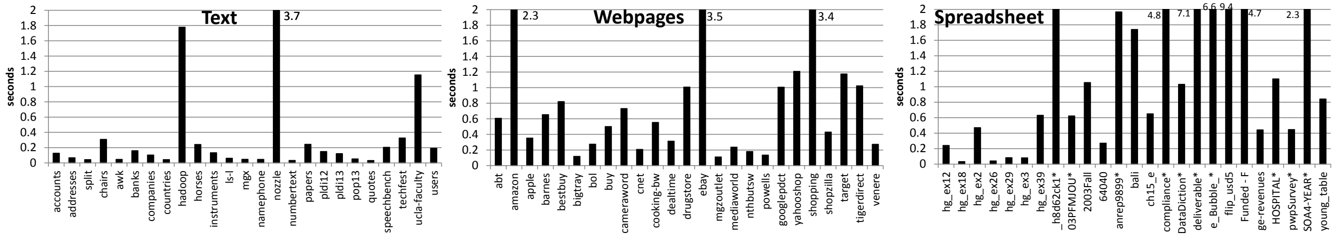

Figure 11: Average learning time of the last interaction across various fields for each document.

Expressiveness Each of the three instantiations of FlashExtract was successfully able to synthesize a desired field extraction program for the various tasks in the respective domains. Thus, FlashExtract supports data extraction DSLs that are expressive enough to describe a variety of real-world data extraction tasks.

Number of Examples FlashExtract required an average of 2.86 examples per field across all documents. Fig. 10 shows the average number of examples (over all fields in the schema for a given document), split by positive/negative instances, for each of the 75 documents in our benchmark. We observe that users have to give more examples to extract data from text files because the structure of text files is more arbitrary. In contrast, webpages and spreadsheets are generally more structured (because of HTML tags and row/column based organization respectively), thus requiring fewer examples.

Synthesis Time Users interactively give more examples to FlashExtract in order to learn a field extraction program. We measure the synthesis time required by FlashExtract during the last iteration before FlashExtract succeeds or fails (since this last iteration has the most number of examples, and thus typically consumes the longest synthesis time). FlashExtract required an average of 0.82 seconds per field across all documents. Fig. 11 reports the average synthesis time (over all fields in the schema for a given document) from the last set of examples, for each of the 75 documents in our benchmark. While most text files and webpages require less than a second per field, spreadsheets sometimes take a few seconds to complete. This is because the spreadsheet DSL is richer with a larger search space.

7. Related Work

We survey some recent work on programming by examples, and then discuss work related to each of our instantiation domains.

DSL Programming by Examples Programming by Example (PBE) techniques often construct a program in an underlying DSL from a set of input/output examples [11]. This area has been gaining renewed interest because recent advances in PBE can help improve the productivity of millions of end-users [12]. For instance, Gulwani *et al.* have developed techniques that enable end-users to perform string transformations [10, 21], number transformations [22], and table transformations [13] from examples. While prior work addresses transformations, we address a different problem of extraction. More significantly, prior synthesis techniques are specialized to an underlying DSL, while ours is more general and can be applied to any DSL that is constructed using our core algebra.

FlashExtract's extraction capability actually complements the transformation capability of prior work; in fact, we have combined them together to provide a better end-to-end user experience. For example, after using FlashExtract to extract data from a text file, the user can perform string transformations [10] or number transformations [22] to modify the extracted fields. Our prototype even allows in-place editing by examples: FlashExtract is used to highlight regions that need to be edited repetitively, and string transformation techniques [10] are used to perform transformation on leaf regions (and these changes are pushed back to the underlying document).

Region Highlighting by Examples LAPIS [18] allows users to highlight sequences of regions in text files and webpages using a variety of means including examples (in addition to user-specified grammars and regular expressions), and then manipulate them using a region algebra. However, LAPIS is limited in its ability to allow users to create and manipulate hierarchical structure by examples. STEPS [24] allows users to color and edit hierarchical regions in text files using small mock examples. In contrast, we allow the user to provide examples on the original input document and not bother about creating mock examples. In particular, we allow the user to provide us a small number of positive instances of the region that they want to highlight on the document. (The user-provided instances need not be a strict prefix of the set of all positive instances. This flexible interaction is enabled by allowing the user to also indicate negative instances).

Besides the user interface, another key difference is how the underlying learning algorithms operate. LAPIS performs lightweight inference (enumerative search over pre-defined patterns) that does not even leverage the document decomposition (when extracting fields with a hierarchical arrangement). STEPS uses machine learning techniques to identify an appropriate composition of a given set of components. In contrast, we perform a systematic search for programs using mostly a divide-and-conquer paradigm where we reduce the problem of learning expressions with a certain top-level operator to the problem of learning sub-expressions based on the properties of that operator. This enables efficient synthesis of larger and more sophisticated field extraction programs.

Data Extraction from Log Files The PADS project [7] has enabled simplification of ad hoc data processing tasks for programmers

by contributing along several dimensions: development of domain specific languages for describing text structure or data format, learning algorithms for automatically inferring such formats [8], and a markup language to allow users to add simple annotations to enable more effective learning of text structure [23] While PADS supports parsing of entire files, FlashExtract allows users to extract only parts of the file thereby avoiding unnecessary complications. PADS's learner only supports a fixed line-by-line chunking strategy to split the records; in contrast, FlashExtract can learn chunking (aka, structure boundaries) from examples, making it suitable for extracting data fields and records that have arbitrary length (and might cross multiple lines). Finally, PADS primarily targets ad hoc text files. Although one can view webpages and spreadsheet as text files, it is unclear if the PADS learning algorithm can be adapted to work effectively for webpages and spreadsheets.

Data Extraction from Webpages Wrappers are procedures to extract data from Internet resources. Wrapper induction is the method to automatically construct wrappers [17]. There has been a wide variety of work in this area, ranging from supervised systems [14, 17, 19], semi-supervised systems [4], to unsupervised systems [5]. FlashExtract differs from the above systems in that its users induce wrappers by interactively giving multiple positive/negative examples. In that sense, FlashExtract is similar to [3]. However, the system in [3] only learns XPath expressions to extract HTML elements. By defining other sequence operators to handle non-HTML text (*i.e.*, text that is within a tag), FlashExtract supports finer grain extraction (*e.g.*, extracting a substring or a sequence of substrings from a text tag, as in Fig. 2). Furthermore, we can leverage advances in wrapper induction research as part of the FlashExtract general framework to support much more sophisticated extraction tasks.

Data Extraction from Spreadsheets Cunha *et al.* [6] detect functional dependencies in spreadsheet data in order to automatically derive even the data schema. However, their technique is not effective over spreadsheets with *hierarchical* data. Abraham *et al.* identify spreadsheet headers automatically [2] and use that to extract relational data. In contrast, FlashExtract extracts (data from) cells based on the properties of the surrounding cells. This allows FlashExtract to deal with spreadsheets with no headers. Furthermore, instead of inferring the whole schema at once, FlashExtract allows users to work in an interactive manner. Users may focus only on cells of interest–this enables robustness on complex spreadsheets.

OpenRefine [1] and Wrangler [16] help users clean and transform their spreadsheet data into relational form. While OpenRefine typically requires users to program, Wrangler automatically infers likely transformation rules and presents them in natural language. However, these tools are limited in their extraction capabilities over spreadsheets with hierarchical data.

8. Conclusion and Future Work

Various common document types such as text files, spreadsheets, and webpages allow users to be creative in using the underlying rich layout capabilities to store multi-dimensional and hierarchical data in a two-dimensional layout. Existing data extraction solutions are domain-specific and require programming skills. We formalize the problem of data extraction in a document independent manner and present an end-user friendly example-based interaction model.

Our synthesis approach advances the state of the art in example based program synthesis. Instead of designing a DSL and a custom synthesis algorithm, we define a core algebra of operators and associate a compositional synthesis strategy with each of them. This provides a free synthesis algorithm for any DSL that is constructed using those operators. We give three examples of DSLs for extracting data from different document types, namely text files, webpages, and spreadsheets. The respective synthesis algorithms are able to extract the intended data using very few examples and in real time.

We foresee two interesting directions for future work. (a) Extending our framework to enable data extraction from more sophisticated document types such as PDF documents and images. (b) Designing a good debugging environment that helps users identify potential discrepancies in what the user intended and the result produced by the synthesized program. This is especially important in two scenarios: (i) when the synthesized program might be executed against other documents with similar formatting, (ii) when the initial set of user-provided examples might be used to re-synthesize a new extraction program in light of formatting changes in the initial document (in fact, the potential for robustness to changes in the input structure might make PBE a more attractive alternative to traditional programming in the first place!)

References

[1] OpenRefine. http://openrefine.org/.

[2] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VL/HCC*, 2004.

[3] T. Anton. Xpath-wrapper induction by generalizing tree traversal patterns. In *LWA*, 2005.

[4] C.-H. Chang and S.-C. Lui. Iepad: information extraction based on pattern discovery. In *WWW*, 2001.

[5] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[6] J. Cunha, J. Saraiva, and J. Visser. From spreadsheets to relational databases and back. In *PEPM*, 2009.

[7] K. Fisher and D. Walker. The pads project: an overview. In *ICDT*, 2011.

[8] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL*, 2008.

[9] C. Frenz, editor. *Pro Perl Parsing*. APress, 2005.

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

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

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

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

[14] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.*, 23(9), 1998.

[15] M. F. Ii and G. Rothermel. The euses spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Workshop on End-User Software Engineering*, 2005.

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

[17] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *IJCAI (1)*, 1997.

[18] R. C. Miller. *Lightweight Structure in Text*. PhD Dissertation, Carnegie Mellon University, 2002.

[19] I. Muslea, S. Minton, and C. A. Knoblock. A hierarchical approach to wrapper induction. In *Agents*, 1999.

[20] E. Oro, M. Ruffolo, and S. Staab. Sxpath: Extending xpath towards spatial querying on web documents. *Proc. VLDB Endow.*, 4(2), 2010.

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

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

[23] Q. Xi and D. Walker. A context-free markup language for semi-structured text. In *PLDI*, pages 221–232, 2010.

[24] 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.