Automatically Extracting Structure and Data from Business Reports

Stephen W. Liddle,[†] Douglas M. Campbell,[‡] and Chad Crawford[†]

[†]School of Accountancy and Information Systems
Marriott School of Management

[‡]Computer Science Department

Brigham Young University, Provo, UT 84602-3087, U.S.A
Email: liddle@byu.edu, Phone: 801-378-8792, Fax: 801-378-5933

Abstract

A considerable amount of clean semistructured data is internally available to companies in the form of business reports. However, business reports are untapped for data mining, data warehousing, and querying because they are not in relational form. Business reports have a regular structure that can be reconstructed. We present algorithms that automatically infer the regular structure underlying business reports and automatically generate wrappers to extract relational data.

Keywords: business reports, report structure, regular expressions, data and information extraction, automatic wrapper generation

1 Introduction

A considerable amount of clean semistructured data is available to companies through internal business reports created during periodic data processing. Business reports provide data for monitoring account balances, inventory levels, transaction status, current production status, etc. Although the subject matter may differ widely, many business reports share a similar structure.

Businesses that employ state-of-the-art techniques capture reports in a Computer Output to Laser Disk (COLD)[1] storage system that is accessible through an enterprise-wide network. COLD systems support queries based on date, title, free-text scanning (as in regular-expression matching), and precomputed indexes whose definitions have been constructed manually at a significant cost in labor and systems-administration/maintenance effort.

Because business reports are an integral part of the business process, when errors are discovered, corrections must be made, and a new version of the report must be issued. Compared to other sources of information, business reports are clean.[2] If this clean data were available in relational form, it could feed a data warehouse.

For various reasons, in some cases important historical and operational data is only available in a COLD system. In other cases, even when such data is available in legacy database systems or file-processing systems, the variety of different data sources and "middleware" access layers can make it difficult to assemble and integrate information from an organization's databases. Since an organization's business reports provide a clean, comprehensive, integrated view of the underlying data of interest, wrappers to extract this data could be less expensive (and are sometimes the only option).

Giving a user finer granularity access to a business report allows more precise queries. If a business report could be automatically decomposed into relational records, then it would not be necessary for a company to have a mediator constructed for each and every one of its data sources. Automatic decomposition would make possible the movement of information from a COLD system into a relational database where data mining and other information tools are available. Alternatively, automatic decomposition would support a direct SQL interface to a COLD system, permitting data mining and queries directly on the COLD archive.

Without automatic decomposition, an end user must develop ad hoc techniques to extract

[1]Actual COLD systems may use other storage technology besides optical disk, for example tape or RAID; we use the term "COLD" to denote any kind of report archive system. Recently this has also been termed "enterprise reporting," but for brevity we use "COLD."

[2]As used in data warehousing, "clean" data is free of errors and redundancy, and is suitable for storing in the warehouse.

information from a business report. For example, she may manually place data from a business report into her spreadsheet. If she receives the business report electronically, she may programmatically transfer the data to a database application such as Access [1] using specialized tools such as awk [3], perl [25], Cambio [8], InfoXtract [6, 15], or Monarch [19]. The difficulties she faces in an ad hoc approach are: the manual specification, the effort to set up a process, the effort to maintain the process, and the acquisition of sufficient programming skills to modify the process. Automatic decomposition eliminates the report-definition specification inherent in manual or programmatic report-based information extraction.

Automated extraction is possible in narrow application domains [12, 13, 14]. However, the techniques for narrow application domains are infeasible for large report bases because ontologies would have to be manually constructed for each different business report. Semi-automatic techniques for wrappers have also been explored [2, 5, 11, 16, 22], but these techniques do not take advantage of the special structural properties of business reports. The project most closely related to ours is NoDoSE [2], which attacks the more general problem of extracting structure from any kind of semistructured document. We apply techniques specific to the business-report domain.

This paper presents a system that utilizes a lattice of field descriptions to automatically identify fields. From field-level descriptions, this system then infers line types that describe the kinds of lines found in a particular business report, and it infers and factors out page headers and footers, yielding a line-type sequence whose regular structure can be inferred using standard algorithms [17, 18]. Our system, implemented in Java, stores extracted information in relational tables according to line type and line-group structure.

The remainder of this paper has four sections. Section 2 gives a high-level system overview. Section 3 gives a detailed description of key algorithms and data structures. Section 4 gives the results of a report survey. Section 5 gives our conclusions.

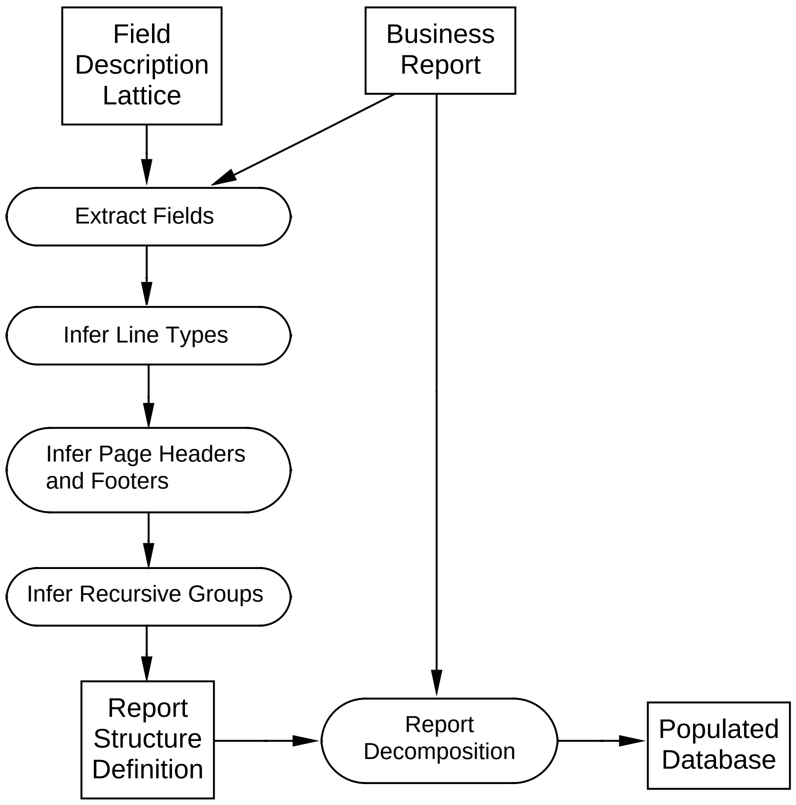

Figure 1: Business report structure and data extraction process.

2 Overview

Figure 1 outlines the two phases of the business report decomposition process: (1) the four steps of report-structure inference, and (2) report decomposition. The input is a business report R, about which we make five assumptions:

1. R is composed of fields that are aggregated into lines, which are in turn aggregated into larger structures.
2. R is in printable ASCII and represents meaningful human-readable information.[3]
3. R uses the ASCII form-feed character (FF) as the page delimiter, and the ASCII line-feed character (LF) as the line delimiter.[4]
4. Each page has the same number of lines R_L, and the width of each line is W characters, padded with blanks if necessary.
5. Blank lines and blanks between fields are for human readability only.

[3]We use ASCII, but EBCDIC or another character set could be used in similar fashion. In the case of EBCDIC, it is easy to translate from EBCDIC to ASCII format. In the case of non-English character sets or non-U.S. business reports, different regular expressions would be required.

[4]There is no difficulty in using CR-LF as a line delimiter as on PC systems.

```
RUN  05/21/99  12:34:56  00551   L A R G E   C D   R E P O R T   ACCR: 04/26/99   POST: 05/21/99   PAGE 001

   CUST NBR    CD NBR             N A M E                       BALANCE          RATE    MATURITY  OFC


   006 9994    10355   JASON MASON CONSTRUCTION INC          100,000.00        .06005   03/07/99
   008 9992     9657   FANNY M RYEBERG                       300,000.56        .05990   04/22/99  MS
   009 9991     9541   JOHN SMITH JR                       1,100,000.00        .05990   04/22/99  MS
   011 9989    11225   BARNEY FIFE                           105,529.23        .06250   05/16/99

                       * * * TOTAL LARGE CD * * *          1,605,529.79
```

Figure 2: A typical type I report page.

```
   CHECK#  AMOUNT   DATE      CHECK#  AMOUNT   DATE      CHECK#  AMOUNT   DATE
    1001    23.99    4/9       1006    13.00   4/15       1011     7.63   4/21
    1002    16.50   4/11       1007     9.99   4/15       1012    16.00   4/25
   *1004    72.11   4/12       1008   155.76   4/15
    1005   145.62   4/13      *1010    10.65   4/17      TOTAL CLEARED:  $471.25
```

Figure 3: A portion of a typical type II report page.

We have observed two major categories of business report structure, distinguished by the relationship between line and record structures. Both kinds of business reports have possible page headers, possible page footers, and a report body that consists of repeating detail lines. A *type I* detail line has columns (fields), belongs to a distinct line-type category, and contains information about a single record. In contrast, a *type II* detail line contains fields pertaining to several records. A type I business report contains only type I detail lines. A type II business report contains type II detail lines (and may also contain type I detail lines).

Figure 2 gives an example of a simple type I report.[5] Each detail line in Figure 2 describes a particular certificate of deposit. Figure 3 shows a portion of a type II checking-account statement. Each detail line in Figure 3 lists two or three cleared-check items, each of which has a check number, amount, and date cleared.

When we correctly identify the basic line types that exist within a type I report, then we can extract the report structure. In contrast, extracting the structure of a type II report requires information beyond line classification. In this paper we discuss type I reports. Type II reports are the subject of a separate paper.

Our process starts with a type I business report R, and a field-description lattice F

[5]None of the data in this paper is actual customer data, but the patterns are based on actual business report structures not designed by us.

(described in Section 3.1), infers the structure of R, stores its definition in a relational database, decomposes R, and stores its decomposition in the database. The contents of R can now be queried. This paper focuses on the four steps of the report-structure inference phase, which consists of the following four steps (corresponding to Algorithms 1 through 4 respectively).

1. For each line t of R, decompose t into its sequence of fields.
2. Infer B, the set of basic line types of R. For each line t of R, assign t its basic line type from B.
3. Infer page headers and footers for R. Factor out the page structure from R's line type description.
4. Infer R's recursive line groups.

This system is implemented in Java 2, using the OROMatcher 1.1 regular-expression library [21] for matching and extracting substrings from lines. We used mySQL [20] for the database management system and twz1jdbcForMysql [23] for the JDBC interface to mySQL. Source code is available on our Web site [10].

2.1 Notation

Before proceeding, we introduce notation and terminology. In general, let $R = \langle R[1], ..., R[R_P] \rangle$[6] be a business report with R_P pages, each with R_L lines, $R[i]$ denoting the i'th page. Each page is a sequence of lines, so $R[i] = \langle R[i][1], ..., R[i][R_L] \rangle$. Each line is a sequence of W characters; after executing Algorithm 1 we can also represent a line as a sequence of fields: $R[i][j] = \langle R[i][j][1], ..., R[i][j][k_{ij}] \rangle$.

Given a line t, we denote a *substring* of t from position j to k, $1 \leq j \leq k \leq W$, by $t[j,k]$. A *field* f in t is a 4-tuple (j,k,i,s), where $s = t[j,k]$ is the substring of t to which f

[6]We always denote an ordered sequence with angle brackets $\langle\ \rangle$. Also, all indexes are 1-based.

corresponds, j is the starting position of f, k is the ending position, and i is a pattern index to be defined in Section 3.1.

Let $f_1 = (j_1, k_1, i_1, s_1)$ and $f_2 = (j_2, k_2, i_2, s_2)$ be fields in lines t_1 and t_2 respectively. We say that f_1 and f_2 *overlap* if there is a q such that $j_1 \leq q \leq k_1$ and $j_2 \leq q \leq k_2$. If f_1 and f_2 overlap, the overlap has one of five *alignment values*:

AGREE	if $j_1 = j_2$ and $k_1 = k_2$ (share left and right endpoints),
ALIGN	they do not AGREE, but both fields are numeric and aligned at the decimal-point position,
LR	they do not AGREE nor ALIGN, but $j_1 = j_2$ or $k_1 = k_2$,
CENTER	they do not AGREE, ALIGN, nor are LR, but $j_1 + \lfloor \frac{k_1 - j_1 + 1}{2} \rfloor = j_2 + \lfloor \frac{k_2 - j_2 + 1}{2} \rfloor$ (center aligned),
OTHER	none of the above apply.

A *field type* f for fields $f_1 = (j_1, k_1, i_1, s_1), ..., f_n = (j_n, k_n, i_n, s_n)$ is a 4-tuple (j, k, i, s) where $j = min(j_1, ..., j_n)$, $k = max(k_1, ..., k_n)$, and i is index of the least upper bound of the elements in the field-description lattice F that are indexed by $i_1, ..., i_n$. F is defined in Section 3.1. For each q, $1 \leq q \leq n$, let s'_q be s_q padded with $j_q - j$ blanks on the left and $k - k_q$ blanks on the right; then $s = \langle s'_1, ..., s'_n \rangle$.

A *line type* t for lines $t_1, ..., t_m$ is a sequence of field types $f_1 = (j_1, k_1, i_1, s_1), ..., f_n = (j_n, k_n, i_n, s_n)$ with two properties: (1) none of the field types may overlap, and (2) $s_1, ..., s_n$ are each ordered sequences containing m strings that correspond respectively to all the fields in lines $t_1, ..., t_m$. By these two properties we guarantee that we can reconstruct the original lines from a line type.

A *group type* d for R is a triple (a, b, c) where a is either a line type or an ordered sequence $\langle d_1, ..., d_n \rangle$ of group types, and b and c are respectively the minimum and maximum number of consecutive occurrences of d observed in R.

3 Structure Extraction Algorithms

As outlined in Section 2, four algorithms extract a business report's structure. Sections 3.1 through 3.4 describe Algorithms 1 through 4 respectively.

3.1 Field Detection

Consider the type I report of Figure 2. The first task is to decompose each line into fields. This is done by applying Algorithm 1 to each line of R.

Let F be the field-description lattice of Figure 4. Indentation in Figure 4 represents precedence, and the universal lower bound is the empty expression (not shown explicitly). Each element of F is a class that describes a set of ASCII strings typically found in business reports. `Julian` is the only class with two immediate successors (`Date` and `Number`). The parenthesized numbers in Figure 4 are used in Section 3.2.1.

Let $E = \langle E[1], ..., E[e] \rangle$ be a sequence of regular expressions corresponding to the field-description lattice of Figure 4 (except for the universal upper bound `Any` and the universal lower bound $\emptyset$). $E[e]$, the last element of the sequence E, has the property that it recognizes any sequence of contiguous non-blank characters ($E[e]$ corresponds to the class `String` in this case). Table 1 in Appendix A shows the regular expressions of E. Notice that no expression $E[i]$ in E matches a string of only blank characters. Algorithm 1 extracts the fields in line t according to E.

Algorithm 1. Extract fields from line.
Input: Regular-expression sequence E and line t.
Output: The sequence of disjoint fields that comprise t relative to E.

```
for $i = 1$ to $e$ do
    while $E[i]$ matches $t$ do
        Set $j$ to the start of the first match.
        Let $k$ be the largest $k \leq W$ such that $E[i]$ recognizes $t[j, k]$.
        Record the field as the 4-tuple $(j, k, i, t[j, k])$.
        Replace the characters of $t[j, k]$ with a special non-ASCII symbol.
    end while
end for
Sort the fields by $j$, the beginning field position.
```

```
Any (1)
     String (.6)
          Time (.3)
               Hour Minute Second (0)
               Hour Minute (0)
          Date (.3)
               Julian (0)
               Day Month Year (0)
               Month Day Year (0)
               Year Month Day (0)
               Month Year (0)
               Month Day (0)
               Day Month (0)
          Phone Number (.3)
               Phone with Area Code (0)
               Phone without Area Code (0)
          ID Code (.3)
               ID Begins with Letters (0)
               ID Ends with Letters (0)
               ID with Digits, Dashes (0)
          Number (.1)
               Julian (0)
               Percent (0)
               Negative (0)
               General Number (0)
               Fraction (0)
               Currency (0)
               Currency with Dollar Sign (0)
          Page Number (0)
          Field Label (0)
          Dividing Line (0)
```

Figure 4: Field-classification lattice.

Regular-expression matching can be linear in the length of the text to be matched (if we accept exponential space in pathological cases) [4], so the inner loop runs in $\mathcal{O}(W)$ time. Since there are e expressions, the outer loop executes e times. Thus, Algorithm 1 executes in $\mathcal{O}(eW)$ time. Since $E[e]$, the last regular expression, always recognizes contiguous non-blank characters, Algorithm 1 terminates and extracts all fields from t. The step that replaces the characters of $t[j,k]$ with a special symbol forces the fields to be disjoint since $t[j,k]$ can no longer be matched by any expression.

3.2 Basic Line-Type Inference

Algorithm 2 is the heart of our technique. It infers basic line types that describe categories of lines in a business report. Before presenting the algorithm we define three field- and line-distance measures.

We first introduce two different distances between fields: a first-order distance, and a second-order distance. First-order distance measures field distance using a character-level string comparison. Second-order distance yields a similarity metric based on the field-classification lattice. A traditional method for characterizing string similarity is *edit distance* [24], which describes the cost of transforming one string into the other. But the computation of edit distance is $\mathcal{O}(mn)$ where m and n are the lengths of the strings being compared. Our simple but adequate first-order distance can be computed in $\mathcal{O}(max(m, n))$ time.

We measure field distances using the minimum of first- and second-order distances together with alignment information (e.g. are the two fields left justified or decimal-aligned). Based on this field distance metric we define a line distance, used in Algorithm 2 to decide when two line types belong to the same cluster.

3.2.1 Field Distance

Let s_1 and s_2 be non-empty ASCII strings. Without loss of generality, we assume that $|s_1| \leq |s_2|$. The *first-order distance* between s_1 and s_2, is:

$$\delta_{string}(s_1, s_2) = \frac{|s_2| - |s_1| + \sum_{i=1}^{|s_1|} \delta_{_K}(s_1[i], s_2[i])}{|s_2|} \tag{1}$$

where $\delta_{_K}(a, b)$ is the Kronecker delta function, namely 1 if $a \neq b$ and 0 if $a = b$.

Let $f_1 = (j_1, k_1, i_1, s_1)$ and $f_2 = (j_2, k_2, i_2, s_2)$ be field types. Recall that s_1 and s_2 are ordered sequences of strings. The *first-order field distance* between f_1 and f_2 is:

$$\delta_1(f_1, f_2) = \sum_{p=1}^{|s_1|} \sum_{q=1}^{|s_2|} \frac{\delta_{string}(s_1[p], s_2[q])}{|s_1| \cdot |s_2|} \tag{2}$$

Our first-order distance uses a (trivial) lattice on characters and ignores the higher-order structure associated with fields. Our second-order distance uses the regular-expression sequence E and field-description lattice F described in Section 3.1. E and F have three important properties:[7]

1. *Lattice.* Each pair of elements in F has a unique least upper bound.

2. *Covering.* Every ASCII string is a member of at least one class in F.

3. *Consistency.* Let $F[i]$ and $F[j]$ be classes in F, and let $E[i]$ and $E[j]$ be the regular expressions in E that correspond to $F[i]$ and $F[j]$, respectively. If $F[i]$ precedes $F[j]$ then the language recognized by $E[i]$ is a subset of the language recognized by $E[j]$.

We define a function ν that assigns each element of F a *value*; more specific classes have lower values than more general classes. Values for ν, in the interval $[0, 1]$, are shown in parentheses in Figure 4, and were determined empirically.

Given these properties, we define the *second-order field distance*. Let $f_1 = (j_1, k_1, i_1, s_1)$ and $f_2 = (j_2, k_2, i_2, s_2)$ be two field types. Let $F[i_1]$ ($F[i_2]$) be the class in F corresponding to the regular expression $E[i_1]$ ($E[i_2]$) that recognizes f_1 (f_2), and let *lub* be the least upper bound of $F[i_1]$ and $F[i_2]$. Without loss of generality, we assume that $\nu(F[i_1]) \leq \nu(F[i_2])$. The *second-order field distance* between f_1 and f_2 is:

$$\delta_2(f_1, f_2) = min(1,\ P \cdot \nu(lub) - \nu(F[i_2])) \tag{3}$$

The difference component of Equation 3 returns a low value for fields whose classes are relatively close. The P term is an empirical constant to penalize fields whose least upper bound is relatively general; we assigned P a value of 1.1 in our experiments. Finally, to ensure that a distance stays in the interval $[0, 1]$, Equation 3 uses the $min(1, ...)$ expression.

[7]These properties require careful construction of the field-description lattice and regular-expression sequence, and we do not formally prove that they hold. For our purposes it is sufficient simply to assume that these properties hold. For the lattice F in Figure 4 the class `Any` is defined to be the set of all ASCII strings. The *consistency* property can be guaranteed if we replace each superior regular expression s by the disjunction of s with each regular expression i that is inferior to s.

Given the first- and second-order field distances of Equations 1 and 3, we define $\delta_{field}(f_1, t_2)$, the *field distance* between field type $f_1 = (j_1, k_1, i_1, s_1)$ in line type t_1 and the sequence of field types of line type t_2, as follows. If f_1 either overlaps no field types of t_2 or overlaps more than one field type of t_2, we define $\delta_{field}(f_1, t_2)$ to be 1. Otherwise, let $f_2 = (j_2, k_2, i_2, s_2)$ be the single field type in t_2 that overlaps f_1, and let $M = min(\delta_1(f_1, f_2),\ \delta_2(f_1, f_2))$. Equation 4 gives the definition of $\delta_{field}(f_1, t_2)$:

$$\delta_{field}(f_1, t_2) = M + (1 - M) \cdot A \tag{4}$$

where A is the alignment value of the overlap of f_1 and f_2, defined in Section 2.1. We determined alignment values empirically, choosing 0, 0, .1, .2, and .4 for AGREE, ALIGN, LR, CENTER, and OTHER, respectively.

3.2.2 Line Distance

Let n_1 be the number of field types in line type t_1, and let n_2 be the number of field types in line type t_2. Based on δ_{field}, we define $\delta_{line}(t_1, t_2)$, the *line distance* between t_1 and t_2. If both n_1 and n_2 are 0, then the value of δ_{line} is defined to be 0. If either n_1 or n_2 is 0 but not both, then the value of δ_{line} is defined to be 1. Otherwise, δ_{line} is defined according to Equation 5:

$$\delta_{line}(t_1, t_2) = \frac{1}{2}\left(\sum_{i=1}^{n_1} \frac{\delta_{field}(t_1[i], t_2)}{n_1} + \sum_{i=1}^{n_2} \frac{\delta_{field}(t_2[i], t_1)}{n_2}\right) \tag{5}$$

3.2.3 Line-Type Inference

Algorithm 2. Infer B, the set of basic line types for report R.
Input: R, after Algorithm 1.
Output: B, the set of basic line types for R, and
L, a mapping from the lines of R to line types in B.

```
Make a copy $Q$ of $R$:
    for each $R[i][j], 1 \le i \le R_P, 1 \le j \le R_L$ do
        Create a new line type $t_1$ for $R[i][j]$.
        if $t_1$ duplicates a line type $t_2$ in $Q$ then
```

```
            Generalize $t_2$ to cover $t_1$.
        else
            Add $t_1$ to $Q$.
        end if
    end for
Reduce line types in $Q$ to $B$:
    Let $B = \emptyset$.
    for each line type $Q[i]$, $1 \leq i \leq |Q|$ do
        Let $m$ be the smallest $\delta_{line}(Q[i], t)$ from $Q[i]$ to any line type $t$ in $B$.
        if $B = \emptyset$ or $m > T$ then
            Add $Q[i]$ to $B$.
        else
            Generalize $t$ to cover $Q[i]$.
        end if
    end for
Construct array $L$ so that $L[i][j]$ is the line type in $B$ that covers line $R[i][j]$.
```

Algorithm 2 terminates in $\mathcal{O}(G \cdot R_{_P} \cdot R_{_L})$ time, where G is the cost of the "generalize" operation, described below. $T = .3$ is a threshold chosen empirically. Line types t_1 and t_2 are duplicates if and only if their associated field-type sequences are identical up to the field text, i.e. (a) they have the same number of field types, (b) the corresponding field types have the same left and right positions, and (c) the corresponding field types are both recognized by the same regular expression.

We now define what it means to generalize a line type t_2 to cover line type t_1. For each field type $f_2 = (j_2, k_2, i_2, s_2)$ in t_2, let m be the number of field types in t_1 that overlap f_2. We denote these m field types as $f_{1,1} = (j_{1,1}, k_{1,1}, i_{1,1}, s_{1,1}), ..., f_{1,m} = (j_{1,m}, k_{1,m}, i_{1,m}, s_{1,m})$. There are three possibilities for m:

1. $m = 0$; do nothing with f_2.
2. $m = 1$; set $j_2 = min(j_2, j_{1,1})$; set $k_2 = max(k_2, k_{1,1})$; set i_2 to the least upper bound of i_2 and $i_{1,1}$; and set s_2 to $s_2 \cup s_{1,m}$, padded with blanks as needed.
3. $m > 1$; set $j_2 = min(j_2, j_{1,1}, ..., j_{1,m})$; set $k_2 = max(k_2, k_{1,1}, ..., k_{1,m})$; set i_2 to e; pad the strings in s_2 with blanks as needed, and add to s_2 the strings from $s_{1,1}, ..., s_{1,m}$, joined and filled/padded with blanks as needed.

If any field type f_1 in t_1 was not overlapped by some field type in t_2, add f_1 to t_2. After modifying t_2, if any field types in t_2 now overlap each other, combine them as described above in step 3.

3.3 Page Header/Footer Inference

A type I business report may have page header and/or a page footer. A *page header* for report R is a sequence of line types that appears at the beginning of each page in R. If line-type sequence $A = \langle a_1, ..., a_h \rangle$ is a page header for R, then $(\forall i, 1 \le i \le R_P)(\forall j, 1 \le j \le h) R[i][j] = a_j$. Similarly, a *page footer* is a sequence of line types $Z = \langle z_1, ..., z_f \rangle$ that appears at the end of each page in R (we assume that a page footer always starts at the same offset from the top of page). To distinguish between report detail and page headers or footers, we require that each non-blank line type $t \in A \cup Z$ have the following properties for each page $R[i]$:

- t does not repeat in $R[i]$ two or more times in immediate succession, and
- t appears only once or twice on any single page $R[i]$.

Algorithm 3. Infer page headers and footers from line types.
Input: Array L from Algorithm 2.
Output: Page-header sequence A, page-footer sequence Z, and line-type sequence $\bar{L}$ with A and Z factored out.

```
Mark non-blank line types that cannot be page header/footer candidates:
    If $L[i][j] = L[i][j+1]$ and $L[i][j]$ is non-blank, then mark both $L[i][j]$ and $L[i][j+1]$.
    If $(\exists j, k, l) j \ne k \ne l$ and $L[i][j] = L[i][k] = L[i][l]$, then mark $L[i][j]$, $L[i][k]$, and $L[i][l]$.
Infer page header:
    Find the largest $h, 0 \le h \le R_L$ such that $(\forall i, j) L[i][1] = L[j][1] \wedge ... \wedge L[i][h] = L[j][h]$.
    Set $A$ to the first $h$ line types of page $R[1]$ ($A$ may be empty).
Infer page footer:
    Find the smallest $f, h \le f \le R_L$ such that $(\forall i, j) L[i][f] = L[j][f] \wedge ... \wedge L[i][R_L] = L[j][R_L]$.
    If such an $f$ exists, set $Z$ to line types $f$ through $R_L$ of page $R[1]$;
        otherwise let $Z$ be the empty sequence and set $f$ to $R_L + 1$.
Reduce $L$ to $\bar{L}$ by removing page structure and blank lines:
    Let $\bar{L}$ be the sequence of line types
        $\langle L[1][h+1], ..., L[1][f-1], \ ..., \ L[R_P][h+1], ..., L[R_P][f-1] \rangle$
    Remove all blank line types that appear in $\bar{L}$.
```

Note that whereas L is a two-dimensional array, $\bar{L}$ has only one dimension. Algorithm 3 terminates in $\mathcal{O}(R_L \cdot R_P)$ time.

3.4 Recursive Group Inference

After the page-specific structure of a business report R has been factored out, we can focus on inferring the structure of R's detail section. Miclet's technique [17, 18] is a reasonable and general way to infer regular structure from a set of example strings. Because of the nature of business reports and the simplifying assumptions this allows, it is possible to infer structure from a single example. Our Algorithm 4 is a variant of Miclet's technique, using different decision heuristics governing when we should reduce a recursive group, and restricted to a single example string (the array $\bar{L}$ of line types).

Business reports created with a report-writer[8] are built up from groups of the form uv^kw, where u is a (possibly empty) group header section, v is a detail section that repeats one or more times, and w is a (possibly empty) group footer section. Each of the u, v, and w sections may themselves be composed of other uv^kw structures. We make three assumptions about the uv^kw structure of line-group types for a business report R reduced by Algorithms 1 to 3 to $\bar{L}$:

1. $k \geq 2$; that is, v appears consecutively somewhere in $\bar{L}$.
2. If group v appears $k \geq 2$ times consecutively in $\bar{L}$, it forms the v^k component of a uv^kw structure (and there is no predetermined upper bound for k). Also, uvw (where $k = 1$) may appear in $\bar{L}$ as long as uv^kw, $k \geq 2$ also appears elsewhere in $\bar{L}$.
3. Groups u, v, and w may not appear in $\bar{L}$ individually (outside of a uv^kw sequence). There are no optional lines in a group. If the real report structure is uv^kw, u and w always appear together with v^k.[9]

We give three examples, representing a line types with lowercase letters. **Example 1.** The sequence $abccc$ is a line group with a group header $u = ab$, a detail section $v^k = c^k$, and an empty group footer, $w = \emptyset$. The reason for this particular uv^kw solution is that c

[8]Most business reports created by custom programming also follow these conventions.

[9]This assumption does not hold for all type I business reports, but we leave such reports for future investigation.

is the only repeating line type in our example. **Example 2.** The sequence *abccabcccc* is $(abc^2)(abc^4)$ which matches $(abc+)(abc+)$. Thus, an expression to describe the structure of such a report is $(abc+)+$. **Example 3.** The sequence *abccbccdabcd* is formed by repeating and nesting. We first create the inner group $e = bc^k$, with header b and detail section c. By substitution, the sequence is now *aeedaed*. Let $f = ae^kd$, with header a, detail section e^k, and footer d. By substitution, the sequence is now ff, which is a group with an empty header and footer, and a detail section f^k. The expression describing this report structure is $(a(bc+) + d)+$.

Essentially, Algorithm 4 reduces the regular expression defined by the line-type sequence $\bar{L}$ to a more compact regular expression G that describes the recursive structure of $\bar{L}$.

Algorithm 4.	Infer recursive line-type groups.
Input:	Basic line-type sequence B from Algorithm 2 and line-type sequence $\bar{L}$ from Algorithm 3.
Output:	Recursive line-group structure.

```
Let $g$ be a set of group types that contains one entry for each line type in $B$.
Map $\bar{L}$ to $G$ by substituting each line type with its corresponding group type from $g$.
changed = **true**
**while** $|G| > 1$ **and** changed = **true do**
    changed = **false**
    **for** i = 1 to $|G|$ **do**
        Find the smallest $j > i$ such that $G[i] = G[j]$ and conditions 1, 2, and 3 hold.
            Let $v = \langle G[i], ..., G[j-1] \rangle$.
            **Condition 1.** Every group type in $v$ is unique.
            **Condition 2.** The sequence $vv$ occurs in $G$.
            **Condition 3.** After substituting a new group type for each occurrence of
                             $v$ in $G$, there is no group type in $v$ that still occurs in $G$.
        **if** $j$ exists **then**
            Create a new group type $x$ whose definition is the sequence $v$.
            Add $x$ to $g$.
            Substitute $x$ in $G$ everywhere $v$ appears.
            changed = **true**
            Replace all consecutive occurrences of $x...x$ in $G$ by a single $x$, and mark $x$
                with the minimum and maximum consecutive-occurrence counts.
        **end if**
    **end for**
**end while**
```

Algorithm 4 is a least-fixed point algorithm, where the fixed point upon which we converge is a regular expression to describe $\bar{L}$. Because $|G|$ is initially $|\bar{L}|$, the **while** loop can execute at most $|\bar{L}|$ times (since we only loop as long as a change has been made, and a change must always reduce the size of G by at least 1). The *for* loop also executes at most $|\bar{L}|$ times. Verifying Conditions 1, 2, and 3 and substituting x for v can both be done in $\bar{L}$ time. Thus, Algorithm 4 executes in $\mathcal{O}(|B| + |\bar{L}|^3)$ time. In practice, Algorithm 4 usually took between one and three passes to converge, running orders of magnitude faster than the worst case just described. (This is because v^k sequences tend to be long.)

4 Results

There are five areas where we used empirically determined values to control the business-report structure and data extraction process: (1) the regular expressions used to recognize fields, (2) the values (ν) associated with each class in the field-description lattice F, (3) the value of alignment constants (AGREE, ALIGN, LR, CENTER, OTHER), (4) the threshold for line-type generalization, and (5) the penalty for least-upper-bound generality in Equation 3. We used hundreds of reports from four different organizations as the basis for our choices.

To test our process, we used 76 business reports from a separate organization that had not been used in the training phase. Of these 76 reports, 7 were not type I. An additional 7 reports were too short to be meaningful (i.e. they comprised a single page containing only page headers or a single detail line). Of the 62 remaining reports, our process correctly extracted the structure and data for 40 reports, but failed with 22. The 22 failures point out directions for future enhancement. We discuss four.

1. E, our sequence of regular expressions for matching fields, was sometimes insufficient. We give four examples. (i) In one case, two fields that were usually separated by a single blank space had a number sign (#) instead of a space on one line. This caused the two fields to be recognized as a single string, which in turn caused the creation of an extra line type that

interfered with the recursive line-type group inference (Algorithm 4). (ii) In another case, we discovered decimal-aligned numeric fields that were left-filled with underscores. Furthermore, these underscores abutted the string field on the left (e.g. “`One____5.52`” and “`Two__934.22`”). Our system recognized the string portion together with the padding underscores as a single field, and the numbers as a second field. Because of the overlapping of these fields, our system generated too many line types for this report. (iii) In another case, we discovered currency amounts specified with 4 digits after the decimal point, rather than the more common 2 digits. Due to the order of our expressions, our system broke such fields in two, which caused too many line types to be generated. (iv) Finally, we found a string field that had two internal spaces (e.g. “`XXXX  XX`”), but our `String` pattern only expects one internal space. This caused the field to be split and an extra line type to be generated. All of these problems can be corrected by tuning E. For our test set, the amount of tuning required would have been small. Adjustments to E are also required for non-U.S. business reports.

2. By far the most common reason for our process to fail was the problem of optional fields in a line type. With more fields present on a line, our distance formulas are more tolerant to optional fields. However it is often the case that lines with few fields also have optional fields, and for lines with many fields, it is also often the case that several fields are optional. Optional fields may lead to our system generating too many line types. Tuning the threshold T of Algorithm 2 for a particular report can sometimes fix this problem, but it is not a general solution.

3. There were several cases where we did not generalize two line types because of the simplistic structure of Algorithm 2, which decides when to generalize based on a threshold. In a future study we will apply clustering techniques such as recursive partitioning or nearest neighbor (as in [9]) to find a better decision function to control when we generalize line types. Such techniques are more likely to be general across business reports with very different line types, and will not be as sensitive to the order of processing line types.

4. Sometimes our uniformity assumption for line-type groups did not hold. That is,

Algorithm 4 assumes that if uv^kw is a line-type group, then u, v^k, and w always appear together. In some cases lines in a uv^kw structure are optional, and in other cases (especially for short lines) a single line type may be reused in two distinct uv^kw structures. Algorithm 4 needs to be revised to accommodate optional lines in a line-type group.

5 Conclusions

It is possible to *automatically* extract structure and data from business reports. Our process correctly extracted the structure and data in 40 out of 62 type I business reports in a test set we had not seen before.

While these initial numbers are encouraging, much work remains to be done. In Section 4 we mentioned four areas needing improvement: (1) field recognition, (2) detecting optional fields, (3) improved line-type clustering techniques, and (4) handling optional lines within a line-type group. We also plan to study structure and data extraction for type II reports. Here it may be possible to use segmentation techniques like those applied in document imaging and optical character recognition (OCR) algorithms (e.g. [7]). This may also enable more accurate extraction of fields from lines, and may shed light on improved techniques for type I line-type clustering. In the current investigation we have assumed fixed-width fields (padded with blanks as needed); since some reports have variable-width fields, our process needs to be extended to accommodate such reports. Also, our assumption that fields are separated by white space does not always hold (some reports are designed to be printed on forms, which may have lines between characters to divide fields). Future work should examine ways to determine field boundaries in the absence of white space.

One weakness of our approach is the number of fixed, empirically determined constants associated with our algorithms. We can surely achieve better results by using adaptive techniques to dynamically compute and adjust these constants whenever possible.

After we have more fully mapped out structure and data extraction for type II reports, we will construct a compressed data structure that contains a full inverted index of the information in a business report R, together with sufficient information to reconstruct the original pages of R from the inverted index. Often it is not enough to merely return the data associated with a particular page; regulatory constraints (at a financial institution, for example) may require that original pages be returned (e.g. records may be subject to subpoena in legal proceedings, in which case the original report pages must be printed). Thus, after fully inverting the data in a report, we must still be able to retrieve the original report pages, including white space. The extensive structural information our system generates constitutes an excellent domain-specific model for compressing reports.

Our business-report structure and data extraction system is implemented in Java. We also implemented a graphical pattern editor tool to assist in the creation and debugging of regular expressions for field extraction. This tool, available from our Web site as PatternEditor 1.0 [10], has general applicability for regular-expression debugging beyond our current project.

References

[1] Microsoft Corporation access page. URL: http://www.microsoft.com/access/.

[2] B. Adelberg. NoDoSE—a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'98*, pages 283–294, Seattle, Washington, June 1998.

[3] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language.* Addison-Wesley, Reading, Massachusetts, 1988.

[4] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[5] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, December 1997.

[6] Bruce Silver Associates. Mining mainframe reports: Intelligent data extraction from print streams, October 1997. URL: http://www.iacorporation.com/assets/press_-releases/InfoXtract_White_Paper.html.

[7] H.S. Baird. Background structure in document images. *International Journal of Pattern Recognition and Artifical Intelligence*, 8(5):1013–1030, 1994.

[8] Data Junction Corporation home page. URL: http://www.datajunction.com/.

[9] R. Chung and K.L. Leung. An iterative clustering algorithm for interpretation of imperfect line drawings. *International Journal of Pattern Recognition and Artifical Intelligence*, 10(8):867–886, 1996.

[10] Data Extraction Group home page. URL: http://www.deg.byu.edu/.

[11] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48, Marina del Ray, California, February 1997.

[12] D.W. Embley, D.M. Campbell, Y.S. Jiang, S.W. Liddle, D.W. Lonsdale, Y.K. Ng, and R.D. Smith. Conceptual-model-based data extraction from multiple-record Web pages. *Data and Knowledge Engineering*, page to appear, November 1999.

[13] D.W. Embley, D.M. Campbell, Y.S. Jiang, S.W. Liddle, Y.K. Ng, D.W. Quass, and R.D. Smith. A conceptual-modeling approach to extracting data from the web. In *Proceedings of the 17th International Conference on Conceptual Modeling, ER'98, Lecture Notes in Computer Science, 1507*, pages 78–91, Singapore, November 1998. Springer Verlag.

[14] D.W. Embley, D.M. Campbell, R.D. Smith, and S.W. Liddle. Ontology-based extraction and structuring of information from data-rich unstructured documents. In *Proceedings of the 1998 ACM CIKM Seventh International Conference on Information and Knowledge Management (CIKM'98)*, pages 52–59, Bethesda, Maryland, November 1998.

[15] IA Corporation home page. URL: http://www.iacorporation.com/.

[16] N. Kushmerick, D.S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI'97*, pages 729–735, Nagoya, Japan, August 1997.

[17] L. Miclet. Regular inference with a tail clustering method. *IEEE Transactions on Systems, Man and Cybernetics*, 9:737–743, 1979.

[18] L. Miclet. *Structural Methods in Pattern Recognition.* North Oxford Academic Publishers Ltd, London, 1986.

[19] DataWatch Corporation home page. URL: http://www.datawatch.com/.

[20] mySQL home page. URL: http://www.mysql.com/.

[21] Savarese.Org home page. URL: http://www.savarese.org/.

[22] S. Soderland. Learning to extract text-based information from the World-Wide Web. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, KDD-97*, pages 251–254, Newport Beach, California, August 1997.

[23] twz1jdbcForMysql home page. URL: http://www.voicenet.com/˜zellert/tjFM/.

[24] R.A. Wagner and M.J. Fisher. The string to string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[25] L. Wall and R.L. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, California, 1991.

PageNumber	\bpage\s*(no\.?\s*)?\d+\b
DividingLine	^\s*([-_=.*]{3,}\s*)+\s*$
HourMinuteSecond	\b{HOUR}{TIMESEP}{MIN}(\2[0-5]\d{AMPM}?|{AMPM})\b
MonthDayYear	\b({MONTH}{DATESEP}{DAY}\5{YEAR}| {MONTH}\s*{DAY},\s+{YEAR})\b
DayMonthYear	\b{DAY}{DATESEP}{MONTH}\2{YEAR}\b
YearMonthDay	\b{YEAR}{DATESEP}{MONTH}\3{DAY}\b
Fraction	\s\.\d+(-|CR)?(?=(\D|$))
Julian	\b{YEAR}{DAYOFYR}(?=([ ]|$))
DollarSign	$\s*{NUM}
PhoneAreaCode	\b{AREACODE}{PHONE}\b
MonthYear	\b{MONTH}{DATESEP2}{YEAR}(?=([ ]|$))
MonthDay	\b{MONTH}{DATESEP2}{DAY}(?=([ ]|$))
DayMonth	\b{DAY}{DATESEP2}{MONTH}(?=([ ]|$))
Negative	\({NUM}\)|(^|[ ])-{NUM}|(\b{NUM}(-|CR)(?=[ ]|))
Currency	\d*(,\d\d\d)*\.\d\d(?=(\D|$))
Percent	-?(\d+(\.\d+)?|\.\d+)%(?=([ ]|$))
PhoneNoAreaCode	\b{PHONE}\b
IDBeginsAlpha	\b[a-z]\w*(-\w+)*-\w*\d\b
IDEndsAlpha	\b\d\w*(-\w+)*-\w*[a-z]\b
IDDigitDash	\b\d+(-\d+)+\b
HourMinute	\b{HOUR}{TIMESEP}{MIN}\b
GeneralNumber	\b{NUM}
FieldLabel	\b\w+(\s\w+)*:(?=\s)
String	[^]+([ ][^]+)*

Table 1: Ordered field recognition pattern definitions.

A Field Pattern Details

The actual expressions used to determine fields are shown in Table 1. These are Perl 5 regular expressions with the addition of our own simple macro substitution mechanism. A macro is a name in curly braces, such as {HOUR}. Table 2 gives the macro definitions. For those who are not familiar with Perl 5 regular expressions, the OROMatcher documentation contains a helpful quick reference [21]. Note that the expressions in Table 1 are listed in order of disambiguation precedence (e.g. PageNumber > DividingLine > FieldLabel > ... > String).

NUM	{NUM1}|{NUM2}|{NUM3}|{NUM4}|{NUM5}|{NUM6}
NUM1	\b\d{4,}\.\d+(?=(\D|$))
NUM2	\b\d{1,3}(,\d\d\d)+\.\d+(?=(\D|$))
NUM3	\b\d{1,3}(,\d\d\d)+(?=(\D|$))
NUM4	\b\d{1,3}\.\d+(?=(\D|$))
NUM5	\.\d+\b
NUM6	\b\d+(?=(\D|$))
MONTH	{STRMONTH}|{NUMMONTH}
STRMONTH	((jan|feb|mar|apr|may|jun|jul|aug|sept?|oct|nov|dec)\.?|january|february|march|april|june|july|august|september|october|november|december)\b
NUMMONTH	0?[1-9]|1[012]
DATESEP	{DATESEP2}|[ ]
DATESEP2	[-/]
DAY	0?[1-9]|[12]\d|3[01]
DAYOFYR	0(0[1-9]|[1-9]\d)|[12]\d\d|3([0-5]\d|6[0-6])
YEAR	\d\d|(1[89]|2[01])\d\d
HOUR	[1-9]|[01]\d|2[0-4]
SEC	{MIN}
MIN	[0-5]\d
AMPM	\s*[aApP]\.?\s*[mM]\.?
PHONE	\d{3}{PHONESEP}\d{4}
AREACODE	(1{PHONESEP})?(([[(]\d{3}[)]({PHONESEP}|[ ])?)|((\d{3}){PHONESEP}))
PHONESEP	[-.]
TIMESEP	[:.]

Table 2: Field pattern macros.