Topes: Reusable Abstractions for Validating Data

Christopher Scaffidi
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
cscaffid@cs.cmu.edu

Brad Myers
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
bam@cs.cmu.edu

Mary Shaw
Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
mary.shaw@cs.cmu.edu

ABSTRACT

Programmers often omit input validation when inputs can appear in many different formats or when validation criteria cannot be precisely specified. To enable validation in these situations, we present a new technique that puts valid inputs into a consistent format and that identifies "questionable" inputs which might be valid or invalid, so that these values can be double-checked by a person or a program. Our technique relies on the concept of a "tope", which is an application-independent abstraction describing how to recognize and transform values in a category of data. We present our definition of topes and describe a development environment that supports the implementation and use of topes. Experiments with web application and spreadsheet data indicate that using our technique improves the accuracy and reusability of validation code and also improves the effectiveness of subsequent data cleaning such as duplicate identification.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *data types and structures, input/output*

General Terms

Reliability, Languages.

Keywords

Data, validation, abstraction.

1. INTRODUCTION

Programs typically do not validate many kinds of human-readable textual input, even though it is well-known that software should validate all inputs [3][14].

One obstacle to more thorough validation is that the two commonly practiced validation approaches, numeric constraints and regular expressions ("regexps") [4][11], are "binary" in that they attempt to differentiate between definitely valid and definitely invalid inputs. Yet for many data categories such as person names and book titles, it is difficult to conclusively determine validity. For instance, no numeric constraint or regexp can definitively distinguish whether an input field contains a valid person name.

A more effective validation approach should not only identify definitely valid and invalid data when feasible, but also identify questionable inputs—values that are not clearly valid but are also not clearly invalid—so those data can receive additional checking from people or programs to ascertain validity. For example, if a textbox contains a questionable person name that has an odd mix of uppercase and lowercase letters, such as "Lincolnshire MCC", the application could ask the user to double-check and confirm the input. Or if the questionable person name was downloaded from a web service, then the application might call a different web service to double-check the value. Each such strategy first requires identifying questionable values.

The second obstacle to more thorough validation is that even valid data from files, web services, and users appear in multiple formats. For example, mailing addresses may specify a full street type such as "Avenue" or an abbreviation such as "Ave.", and books may be referenced by title or ISBN. Prior to using data for printing reports or performing analyses, an application typically must put data into a consistent format. Even if a programmer could write a regexp (with many disjunctions) to recognize multi-format data, the regexp still would leave inputs in multiple formats at the end of validation. Ideally, an abstraction that supports validation should help with transforming data into the format needed by the main application.

Based on these considerations, we present a technique for recognizing questionable data and putting data into a consistent format. Our technique relies on a new abstraction called a "tope", which robustly describes a data category independently of any particular software application and which is reusable across applications and across software development platforms. A tope contains multiple explicitly distinguished formats that recognize valid inputs on a non-binary scale, and it contains transformation functions to map values from one format to another. In comparison, a regexp with disjunctions may recognize several formats, but they are not explicitly distinguished, matching is binary, and no transformations are provided.

To help software engineers implement topes, we have created a Tope Development Environment (TDE). This environment is based on a direct manipulation user interface that we evaluated in previous work, showing that it enables people with minimal formal training in programming to describe data formats quickly and accurately [32].

This paper does not focus on the user interface. Rather, this paper's contributions are the new topes data abstraction that software engineers implement through the TDE, as well as a validation technique that associates topes with input fields in order to identify invalid data and put valid data into a consistent format.

Our empirical evaluation shows that compared to single-format binary validation, our technique has three benefits. First, it increases the accuracy of validation. Second, it increases the reusability of validation code. Finally, putting data into a consistent format improves the effectiveness of subsequent data cleaning, such as identification of duplicate values.

This paper is organized as follows. Section 2 presents real examples of data validation in practice, and Section 3 identifies key characteristics of the data to be validated. Section 4 presents topes in detail and briefly describes the TDE. Section 5 reports on our quantitative evaluation, Section 6 discusses related work, and Section 7 concludes with limitations and future work.

2. MOTIVATING EXAMPLES

Programmers often omit validation for input fields, causing programs to accept invalid as well as inconsistently formatted data. We found many examples of unvalidated fields in "person locator" web applications created in the aftermath of Hurricane Katrina [30].

To assist the search for survivors displaced by the hurricane, numerous teams created web applications so users could store and retrieve data about the status and location of people. Unfortunately, these programmers had no way to know that other teams simultaneously created essentially identical sites. Within a few days, another team recognized that the proliferation of sites forced users to perform substantial repetitive work, so this team created "screen scraper" software to read and aggregate the other sites' data into a single site. We interviewed six programmers, including two members of the aggregator team. One programmer gave us his application's source code, and we inspected the other sites' browser-side code. See [30] for sampling and data analysis methodology.

Each web application provided forms for users to enter data about survivors, and prior to storing inputs in a database, applications validated some inputs by testing them with constraints and regexps. For example, one form had fields for the date and time when a missing person was found. To validate the time, the application checked the string against a regexp. If the regexp check succeeded, then the code checked the hour and minute against numeric constraints. Similar code validated the date.

However, interviewees' web forms did not carefully validate most data. For example, they simply checked if person names were non-empty but did not check if inputs were excessively long or contained strange punctuation (as would be present in an SQL-injection attack [14]). Most fields were not validated at all.

Interviewees explained that they intentionally omitted validation in order to provide end users with maximal flexibility. While they conceded that this resulted in accepting some invalid data, they emphasized that aggressive validation might have prevented many people from entering valid data.

Unfortunately, the lack of validation led to subtle errors. For example, one end user put "12 Years old" into an "address" field on one site. In addition, data formats varied between sites and even within each site. For example, one site used the date format "09.04.2005", while another used "9/04/2005," and another used "2005-09-04". Scrapers recognized this special case and transformed each to "2005-09-04" before aggregation. However, other data categories such as mailing addresses also varied in format, and scraper developers generally did not write code to put these into a consistent format. Because the aggregated data appeared in multiple formats, it was often difficult to determine if entries on different sites or within the same site referred to the same person.

We have found similar examples of unvalidated fields in many other web applications, such as Google Base. Conceptually, Google Base is a large online database, with 13 primary web forms for inserting rows into database tables. Each form corresponds to a table, and each input field in each form corresponds to a table column. However, these conceptual table columns are not static, as users can omit fields/columns or add custom fields/columns on a per-record basis.

For example, the Jobs web form lets users enter job openings. By default, the form displays fields for the job type, industry, function, employer, education level, salary, immigration status, salary type, publish date, description, and location. The web form offers a list of suggested values for many fields (such as "Contractor", "Full time", and "Part time" for the job type), but users are free to type arbitrary values for each field.

Most Google Base form fields accept unvalidated text. That is, Google has not implemented validation for most default fields/columns, including job type, industry, function, employer, or education level. Moreover, even for numeric fields, the forms accept unreasonable numbers (such as a salary of "-45"). When a user adds a custom field/column, no validation occurs.

Web applications are not the only software requiring data validation. For example, among people with minimal formal training in programming, spreadsheets are the most common platform for implementing computations and generating reports [33]. Prompted by the high error rate in spreadsheets [19], researchers have provided techniques for validating formulas and numeric data [6][27].

Yet in one study, nearly 40% of spreadsheet cells contained non-numeric, non-date textual data [10]. In a contextual inquiry, we found that information workers commonly use spreadsheets to gather and organize textual data in preparation for creating reports [34]. Our results were consistent with a previous study which found that nearly 70% of spreadsheets were created for reporting purposes [12]. Despite the importance of textual data, spreadsheets offer no support for validating strings. Though Excel lets users associate data categories such as "social security number" with cells for formatting, this does not actually check the data or display error messages for invalid values. Our validation technique and its supporting tope abstraction are platform-agnostic, applying in particular to spreadsheets and web applications.

3. CHARACTERISTICS OF TEXTUAL DATA

A successful technique must accommodate several key characteristics of the textual data to be validated.

First, the data are human-readable character strings appearing in input fields, such as spreadsheet cells and web form textboxes.

Second, each application's problem domain calls for most fields to each contain values from a certain category, such as company names or mailing addresses. In many categories, a value is valid if it refers to what information extraction researchers call a "named entity"—an object in the real world such as a company or a building [7][16]. Other data categories, such as salaries or temperatures, do not refer to physical entities. In these categories, values must be consistent with categorical constraints implicit in the programmer's conception of the data. For example, salaries should be positive and, depending on the job, generally within some range.

Third, many data categories lack a formal specification. For example, the Jobs web form in Google Base includes an employer field, which usually refers to a company (a named entity). Yet what constitutes a "real" company and, therefore, a valid input? A company does not need to be listed on a stock exchange or even incorporated in order to be real. Even a private individual can employ another person. The category's boundaries are ambiguous.

This leads to the first limitation of constraints, regexps, context-free grammars, types, and similar binary validation approaches [5][17][18][20], which is that they only attempt to identify definitely valid and definitely invalid data. Some systems, such as those that validate data by checking dimensional units, not only are restricted to binary validation but also cannot validate named entities [2][8][9][15]. All of these approaches are unsuitable for data categories with ambiguous boundaries. In contrast, this paper presents a new technique to identify questionable values—those that are not definitely valid but are also not definitely invalid. This technique can be used to ferret out values that deserve double-checking in order to gain more confidence in the validity of data.

Finally, returning to characterizing the textual data in applications, we note that values in many data categories can be "equivalent" but written in multiple formats. By this, we mean that, for data categories referencing named entities, two strings with different character sequences can refer to the same entity. For example, "Google", "GOOG", and "Google, Inc." refer to the same entity, though they have different character sequences. The first exemplifies the company's common name, the second refers to the company by its stock symbol, and the third refers to the company by its official legal name. Within a certain context, people may use other formats to reference companies, such as the TIN identifiers used by the United States IRS.

For categories that do not reference named entities, two strings with different character sequences may be equivalent according to that category's meaning. For example, a temperature of "212° F" is equivalent to "100° C". As with named entities, certain people may use other formats to reference the same temperature, such as "313.15 kelvin" used by physicists.

Because textual data can appear in many formats, inputs are not always in an application's preferred format. For example, when Hurricane Katrina screen scrapers downloaded data from web sites, the data had a mixture of formats. Our validation technique not only helps identify questionable values, but it also helps automate the task of putting data into a consistent format.

4. DATA VALIDATION WITH TOPES

Our approach models each data category as an abstraction called a "tope". By "abstraction", we mean the following:

> The essence of abstraction is recognizing a pattern, naming and defining it, analyzing it, finding ways to specify it, and providing some way to invoke the pattern by its name [35].

Creating an abstraction for a data category involves recognizing the conceptual pattern implicit in that category's valid values, naming the pattern, carefully defining and evaluating the pattern, and ultimately implementing software that encodes the pattern and uses it to validate values. (Here, "pattern" has the dictionary definition of "a consistent or characteristic arrangement" rather than the specific technical meaning of a Design Pattern)

For example, consider a simple data category, email addresses. A valid email address is a username, followed by an @ symbol and a hostname. At the simplest level, the username and hostname can contain alphanumeric characters, periods, underscores, and certain other characters. This abstraction—a pattern that succinctly describes the category—could be implemented in various ways, such as a regexp, a context-free grammar (CFG), or a Java program.

More sophisticated abstractions for email addresses are possible. For example, a more accurate pattern would note that neither the username nor the hostname can contain two adjacent periods. A still more accurate abstraction would note that the hostname ends with ".com", ".edu", country codes, or certain other strings.

Some data categories mentioned in Section 3 are most easily described in terms of *several* patterns. For example, companies can be referenced by common name, legal name, or stock ticker symbol. The common names are typically one to three words, sometimes containing apostrophes, ampersands, or hyphens. The legal names may be somewhat longer, though rarely more than 100 characters, and they sometimes contain periods, commas, and certain other characters. The ticker symbols are one to five uppercase letters, sometimes followed by a period and one or two letters; more precisely, these symbols are drawn from a finite set of officially registered symbols. These three patterns *together* compose an abstraction describing how to recognize companies.

We refer to each such abstraction as a "tope", the Greek word for "place", because each data category has a natural *place* in the problem domain. That is, problem domains involve email addresses and salaries, rather than strings and floats, and it is the problem domain that governs whether a string is valid.

In our data validation technique, a programmer uses our Tope Development Environment (TDE) to implement a tope for a data category. The tope implementation contains two kinds of functions. One kind of function uses grammars to recognize valid values and to parse values into trees, while the other kind of function transforms values between formats by accessing and recombining nodes in parse trees. Plug-ins for popular software development platforms help programmers to associate a tope implementation with a field in an application. At runtime, the application passes values from the field into the respective tope implementation functions to validate and transform data.

The remainder of this section discusses topes in detail and reviews key features of the TDE.

4.1 Topes

The purpose of topes is to validate strings. So we begin with strings and build up to a definition of a tope.

Each string is a sequence of symbols drawn from a finite alphabet $\sum$, so each string $s \in \sum^*$. (In practice, $\sum$ is UniCode.) A data category D is a set of valid strings.

Each tope τ is a directed graph (F, T) that has a function associated with each vertex and edge. For each vertex $f \in F$, called a "format", `isa`$_f$ is a fuzzy set membership function `isa`$_f$:$\sum^* \rightarrow [0,1]$. For each edge $(x,y) \in T$, called a "transformation", `trf`$_{xy}$ is a function `isa`$_t$:$\sum^* \rightarrow \sum^*$. Each vertex's `isa` function recognizes instances of a conceptual pattern. Each edge's `trf` function converts strings between formats.

Each $\texttt{isa}_f$ defines a fuzzy set [36], in that $\texttt{isa}_f$(s) indicates the degree to which s is an instance of one conceptual pattern. For example, when recognizing strings as company common names based whether they have one to three words, a string s with one word would match, so $\texttt{isa}_f$(s) = 1. The conceptual pattern might allow an input s' with four words, but with a low degree of membership, such as $\texttt{isa}_f$(s') = 0.1. The pattern might completely disallow an input s'' with 30 words, so $\texttt{isa}_f$(s'') = 0.

For each tope τ = (F, T), we define the "validation function"

$$\phi(s) = \max_{f \in F}[\texttt{isa}_f(s)]$$

If ϕ(s) = 1, then τ labels s as valid. If ϕ(s) = 0, then τ labels s as invalid. If $0 < \phi(s) < 1$, then τ labels s as questionable, meaning that it may or may not be valid. The ultimate goal of validation is to prevent invalid data from entering the system; for this purpose, τ would be perfectly accurate if $\forall$ s$\in\sum^*$, (ϕ(s) = 0) <=> (s$\notin$D).

Each $\texttt{trf}_{xy}$ converts strings from format x into equivalent strings in format y. For example, the tope in Figure 1 may use lookup tables to match common names and official titles with stock symbols.

Transformations can be chained, so in practice, topes are typically not complete (each vertex directly joined to every other vertex), so the number of functions grows linearly with respect to the number of formats. Some topes may have a central format connected to each other format in a star-like structure, which would be appropriate when one format can be identified as a canonical format. Alternate topologies are appropriate in other cases, such as when a tope is conveniently expressed as a chain of sequentially refined formats.

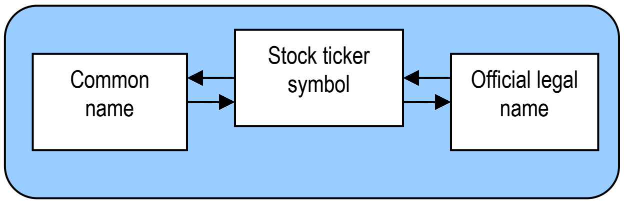


Figure 1: Notional depiction of a simple company tope, with boxes showing formats and arrows showing transformations

4.2 Common Patterns

By reviewing spreadsheets and web forms, we have found that a few specific kinds of patterns describe many formats.

Enumeration patterns: Values in many data categories refer to named entities, of which only a finite number actually exist. Each such category could be described as a tope with a single format whose membership function $\texttt{isa}_f$(s) = 1 if s$\in\Delta$, where Δ is a set of names of existing entities, and $\texttt{isa}_f$(s) = 0 otherwise. If two strings can refer to the same named entity, it may be convenient to partition this set, creating multiple formats each recognizing values in a partition. The $\texttt{trf}$ functions could use a lookup table to match strings in different formats.

For example, Canada currently has ten provinces. A suitable tope for this category would have a format x with

$\texttt{isa}_x$(s) = 1 if s$\in${"Alberta", "British Columbia",…}, 0 otherwise

Depending on the programmer's purpose for this category, a slightly more sophisticated format could also recognize the three territories of Canada, perhaps with

$\texttt{isa}_x$(s) = 0.9 when s references a territory.

The tope could include a second format y that recognizes standard postal abbreviations for province names using

$\texttt{isa}_y$(s) = 1 if s$\in${"AB", "BC"…}, 0 otherwise

The $\texttt{trf}$ functions could use a lookup table to map values from one format to the other.

Similar topes could be used to describe many other data categories, including states in the United States, countries of the world, months of the year, products in a company's catalog, and genders. Each of these data categories contains a small number of values.

List-of-words patterns: Some categories have a finite list of valid values but are infeasible to enumerate, either because the list is too large, too rapidly changing, or too decentralized in that each person might be unaware of valid values known to other people. Examples include book titles and company names. Though a tope could contain a format x with $\texttt{isa}_x$(s) = 1 if s$\in\Delta$ (an enumeration pattern), this format would omit valid values.

Most such data categories contain values that are lists of words delimited by spaces and punctuation. Valid values typically must have a limited length (such as one or two words of several characters each), and they may contain only certain punctuation or digits. Typical lengths and characters vary by category.

A tope could contain a format y with $\texttt{isa}_y$(s) = r$\in$(0,1) if s has an appropriate length and contains appropriate characters. For a tope with an enumeration format x and a list-of-words format y, ϕ(s) = $\texttt{isa}_x$(s) = 1 if s$\in\Delta$, otherwise ϕ(s) = $\texttt{isa}_y$(s) = r if s matches the list-of-words format's pattern, and 0 otherwise.

The tope could omit $\texttt{trf}$ functions connecting formats x and y. However, from an implementation standpoint, the list-of-words format could be generated automatically by examining values in Δ and inferring a pattern using an algorithm such as one provided by the TDE [28]. Conversely, at runtime, if a string s appears often, then it could be added automatically or semi-automatically to Δ.

Numeric patterns: Some data categories contain string representations of numeric data that conform to constraints. Examples include salaries, weather temperatures, and bicycle tire sizes. In many cases, a dimensional unit is present or otherwise implicit, with possible unit conversions. For example, "68° F" is equivalent to "20° C". Weather temperatures less than "-30° F" or greater than "120° F" are unlikely to be valid.

A tope having one format for each possible unit could describe such a category. For each format f, $\texttt{isa}_f$(s) = 1 if s contains a numeric part followed by a certain unit such as "° F", and if the numeric part meets an arithmetic constraint. If s comes close to meeting the numeric constraint, then $\texttt{isa}$(s) = r$\in$(0,1).

For these data categories, converting values between formats usually involves multiplication. Thus, each $\texttt{trf}_{xy}$ could multiply the string's numeric value by a constant, then change the unit label to the unit label required for the destination format y.

Some data categories are numeric but have an implicit unit or no unit at all. An example is a North American area code, which must be between 200 and 999, and which cannot end with 11. For such categories, a suitable tope could have one format.

Hierarchical patterns: Values in many data categories contain substrings drawn from other data categories. One example is American mailing addresses, which contain an address line, a city, a state, and a zip code. An address line such as "1000 N. Main St.

NW, Apt. 110" is also hierarchical, containing a street number (recognizable with a numeric pattern), a predirectional (an enumeration pattern), a street name (a list-of-words pattern), a postdirectional, a secondary unit designator (an enumeration pattern), and a secondary unit (a numeric pattern). Some parts are optional. In most cases, spaces, commas, or other punctuation serve as separators between adjacent parts. Other examples of hierarchical data include phone numbers, dates, email addresses, and URLs.

A format x for a hierarchical pattern could have

$$\texttt{isa}_x(s) = \prod_i \phi_i(s_i) \prod_j \chi_j(s)$$

Here, i ranges over the value's parts, each s_i is the substring for part i (which should be valid according to tope τ_i), and $\phi_i(s_i)$ is the validation function for τ_i; j ranges over "linking constraints" that involve multiple parts, and $\chi_j(s) \in [0,1]$ applies a penalty if any linking constraint is violated. As an example of a linking constraint, in a date, the set of allowable days depends on the month and the year. If s violates this constraint, $\texttt{isa}_x(s) = \chi_j(s) = 0$.

A more specialized $\texttt{isa}_x$ might require that each part i matches a *particular* format $f_i \in \tau_i$ (using $\texttt{isa}_{fi}$ instead of ϕ_i above). Also, $\texttt{isa}_x(s)$ might equal 1 only if s used certain separators (appending additional factors after the χ_j factors). For example, a date format x could accept "12/31/07" but not "Dec 31, 2007", which could instead be accepted by another specialized format y. Transforming "12/31/07" from x to y could apply a `trf` from τ_{month} to the month, applying a `trf` from τ_{year} to the year, and changing the separators.

However, using specialized `isa` functions can lead to combinatorial explosion, since for many categories recognized with hierarchical patterns, each part can take on several formats, and each separator can be selected from a few choices. Section 4.5 addresses this issue.

4.3 Tope Implementations

Just as an abstract type is not executable, topes are not directly executable but must be implemented.

For some simple patterns, software engineers could use a regexp to implement a format in which `isa`(s) = 1 if s matches the regexp, and 0 otherwise. On programming platforms where regexps support capture groups (assigning a name to parts of a value) [11], `trf` functions could be implemented in code that reads a capture group's value and then performs any necessary transformations.

However, as described earlier, regexps are insufficient for identifying questionable values, so the TDE provides more sophisticated mechanisms for implementing topes.

To implement an `isa` function, a programmer uses the TDE's format editor to describe data as a sequence of named parts, with facts specified about the parts (Figure 2). Facts can be specified as "always", "almost always", "often", or "never" true. The user interface has an advanced mode where programmers can enter facts that "link" parts, such as the intricate rules for validating dates.

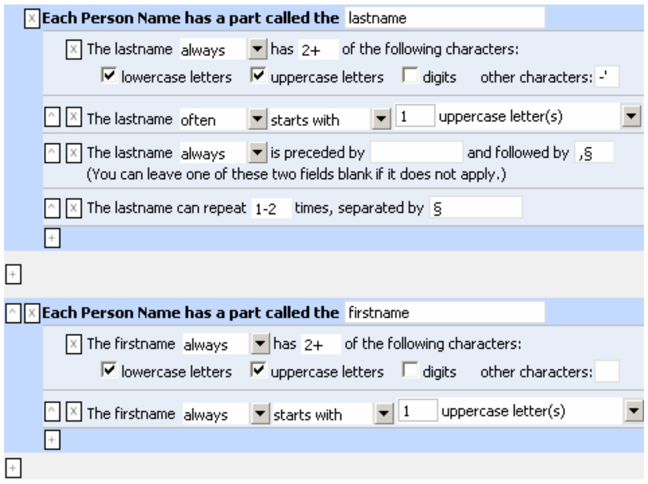
Even information workers, engineering students, and other "end-user programmers" with minimal formal training in software engineering are able to use the TDE's format editor to quickly and correctly describe phone numbers, mailing addresses, and other data [32].

The format editor's main visual feature is its sentence-like prompts in constraint-like facts. For example, facts can specify that a part of the format should be in a numeric range, or that the part should match some other tope (or a specific format in a tope). One somewhat unintuitive aspect of the user interface is that it makes spaces visible by representing them with a special symbol, §. This slightly reduces readability but is preferable to having spaces that programmers cannot see or debug.

From a format description, the TDE automatically generates a context-free grammar with constraints on grammar productions. Using this grammar, the TDE automatically implements an `isa` function that parses inputs with the grammar and tests the parse tree's nodes with the production constraints. The function returns 0 on parse failure, 1 for valid parses that satisfy all constraints, and a number between 0 and 1 if the parse succeeds but violates constraints. The precise return value depends on how many constraints are violated and whether violated constraints should always or often be true [31].

Again using an editor based on sentence-like prompts, a programmer implements a `trf` function as a series of instructions that read text from nodes in a string's parse tree, modify the text, and concatenate the text to form the `trf`'s return value. For example, transforming a person name from "Lastname, Firstname" format to "FIRSTNAME LASTNAME" format requires capitalizing the two parts, permuting them, and changing the separator from ", " to " ".

The primitives supported by the TDE have sufficed for implementing a wide range of topes, including all of the common patterns described in Section 4.2 and all the topes in Section 5.



Each Person Name has a part called the lastname

The lastname always has 2+ of the following characters:
[✓ lowercase letters] [✓ uppercase letters] digits other characters: -'

The lastname often starts with 1 uppercase letter(s)

The lastname always is preceded by and followed by ,§
(You can leave one of these two fields blank if it does not apply.)

The lastname can repeat 1-2 times, separated by §

Each Person Name has a part called the firstname

The firstname always has 2+ of the following characters:
[✓ lowercase letters] [✓ uppercase letters] digits other characters:

The firstname always starts with 1 uppercase letter(s)

Figure 2: Using the TDE to describe person names in "Lastname, Firstname" format

4.4 Validating Data with Topes

The `isa` and `trf` function implementations are stored together in an XML tope implementation file. In order to help software engineers use tope implementations to validate data, the TDE provides plug-ins for several software development tools, including Visual Studio.NET (a web application design tool) and Microsoft Excel.

To validate a field, such as a textbox in a web form or a cell in a spreadsheet, the software engineer uses the tool plug-in to select a tope implementation file and a preferred format x in the tope.

The Visual Studio plug-in generates JavaScript code so that at runtime, the application uses the tope implementation to calculate $\texttt{isa}_x(s)$. If $\texttt{isa}_x(s) = 1$, the form accepts s, and if $\texttt{isa}_x(s) = 0$, the form rejects s. If $0 < \mathtt{isa}_x(s) < 1$, the application displays a warning message so the end user can double-check and possibly correct s before it is accepted. (The software engineer can also specify alternate settings, such as always rejecting any input with $\mathtt{isa}_x(s) < 1$, thus dispensing with the warning message.)

As mentioned above, the software engineer selects a preferred format x, but an input s may match a non-preferred format y better than x. That is, $\mathtt{isa}_y(s) > \mathtt{isa}_x(s) > 0$. In this case, prior to displaying warning or error messages, the generated code automatically uses `trf` functions to transform s into format x. This may require executing a series of `trf`s to traverse the format graph. The present heuristic chooses a shortest path (that is, a minimal number of `trf`s); we may evaluate more sophisticated heuristics in the future. After transformation, the new string is checked with $\mathtt{isa}_x$, since $\mathtt{isa}_x(\mathtt{trf}_{yx}(s))$ could still be less than 1. Because transformation precedes form submission, the user can review the transformed value and correct it if appropriate.

In a spreadsheet, tope implementation files are associated with cells. In this case, no code generation is required. Instead, an interpreter in the plug-in directly uses the tope implementation to validate and transform values. The user interface allows users to ignore warning messages and correct transformations.

4.5 Reuse

As we will show in Section 5, many tope implementations can be reused without modification in multiple applications, both within a development platform (such as web applications) and across platforms (such as reusing a tope implementation in a web application even though it was originally created for a spreadsheet).

However, different software applications may require slightly different validation for a particular data category, thus limiting reusability. For example, even though most academic course codes have a 3-letter department abbreviation, then a hyphen, and then a 3-digit course number, specific courses differ by institution. Although this simple abstraction covers most institutions' courses, it does not take advantage of institution-specific information that could improve accuracy. (These tradeoffs are analogous to typical tradeoffs in deciding whether to reuse a general-purpose component in a new application or to create a specialized component.)

To maximize a tope's reusability, we have found that it is often useful to include a general-purpose format that accepts a variety of valid values and then add specialized formats as needed. For example, a general-purpose American phone format might accept "412-555-1212", "(412) 555-1212", and "412.555.1212". When validating data in a particular application, we add a specialized format such as "###-###-###" and specify it to be the preferred format, "feeding" it from the general-purpose format with a transformation. Implementing the specialized format and transformation function typically requires only a few minutes in the TDE. At runtime, our plug-ins automatically uses our transformation to convert inputs such as "412.555.1212" to the preferred format.

In our experience, this strategy prevents the combinatorial explosion of hierarchical formats mentioned in Section 4.2. The reason is that we only need to implement one general-purpose format as well as the specialized formats that are actually used as preferred formats in real applications, rather than having to create a specialized format for each hypothetically possible combination of parts and separators.

5. EVALUATION

We tested topes using the 720 spreadsheets in the EUSES Spreadsheet Corpus's "database" section, which contains a high concentration of string data [10]. We analyzed the data to classify values into categories. After inspecting the data, we used the TDE to implement topes for the 32 most common categories.

To evaluate how well these topes classified data as valid or invalid, we randomly selected test values for each category, manually determined the validity of test values, and computed topes' accuracy. We judge accuracy using F_1, a standard machine learning statistic used to evaluate classifiers such as document classifiers and named entity classifiers, with typical F_1 scores in the range 0.7 to 1.0 [7][16].

$$Recall = \frac{\text{\# of invalid inputs successfully classified as invalid}}{\text{total \# of invalid inputs}}$$

$$Precision = \frac{\text{\# of invalid inputs successfully classified as invalid}}{\text{\# of inputs classified as invalid}}$$

$$\mathrm{F}_1 = \frac{2 \cdot \mathrm{Recall} \cdot \mathrm{Precision}}{\mathrm{Recall} + \mathrm{Precision}}$$

To evaluate reusability, we tested whether web application data could be accurately validated by the topes that we implemented based on spreadsheet data. We extracted data from the Google Base web application and one Hurricane Katrina web site and identified data categories where our tope implementations could be reused. For each category, we randomly selected test values, manually determined their validity, and then computed F_1.

Finally, we focused on the problem that Hurricane Katrina data often had mixtures of formats, which interfered with identifying duplicate values. To evaluate if topes could help with this problem, we implemented and applied transformations to test data, then counted the number of duplicate values before and after this transformation to consistent formats.

5.1 Data Categories in the Test Data

Each spreadsheet column in the EUSES corpus typically contains values from one category, so columns were our unit of analysis for identifying data categories. To focus our evaluation on string data, we only extracted columns that contained at least 20 string cells (i.e.: cells that Excel did not represent as a date or number), yielding 4250 columns. We clustered these with hierarchical agglomerative clustering [7]. For this algorithm, we used a between-column similarity measure based on a weighted combination of five features: exact match of the column's first cell (which is typically a label), case-insensitive match of the first cell, words in common within the first cell (after word stemming [22]), values in common within cells other than the first cell, and "signatures" in common. Our signatures algorithm replaced lowercase letters with 'a', uppercase with 'A', and digits with '0', then collapsed runs of duplicate characters [28].

The algorithm yielded 562 clusters containing at least 2 columns each, for a total of 2598 columns. We inspected these clusters, examining spreadsheets as needed to understand each column's data. In many cases, we found that clusters could be further merged into larger clusters. For example, the columns with headers "cognome", "faculty", and "author" all contained person last names.

Of the 2598 columns covered by clusters, we discarded 531 (20%) that did not refer to named entities nor seemed to obey any implicit constraints; this was generic "text". In addition, we discarded 196 columns (8%) containing database data dictionary information, such as variable names and variable types. These data appeared to be automatic extracts from database systems, rather than user-entered data, and seemed too clean to be a reasonable test of topes. We discarded 76 columns (3%) containing text labels—that is, literal strings used to help people interpret the "real" data in the spreadsheet. Finally, we discarded 82 columns (3%) because their meaning was unclear, so we were unable to cluster them. We discarded a total of 885 columns, leaving 1713 columns (66%) in 246 clusters, each representing a data category.

The Hurricane Katrina web application had 5 unvalidated text fields: person last name, first name, phone, address line, and city. (We omitted a dropdown field for selecting a state from our test data.)

Google Base had 13 main web forms, with 3 to 15 fields each. We manually grouped the 66 unvalidated text fields into 42 categories, such as person, organization, and education level.

5.2 Coverage of Data Categories

In many situations, when categorizing items into groups, a few groups cover most items, followed by a tail of groups each containing few items. One significant question is whether that tail is "light" (a few groups cover most items) or "heavy" (many items are in small groups) [1].

We plotted the number of spreadsheet columns per category and performed non-linear fits to determine if the distribution was heavy or light. A heavy-tail power-law distribution fit the data better than a light-tail exponential distribution ($R^2 = 0.97$ versus 0.80, respectively), indicating that many special-purpose topes exist. This suggests a need for reusable abstractions (since some categories occur frequently) as well as a TDE to support implementation of diverse custom topes, rather than just a closed library of reusable topes.

The most common 32 categories covered 70% of the spreadsheet columns. These included Booleans (such as "yes" or "x" to indicate true, and "no" or blank to indicate false), countries, organizations, and person last names. The tail outside of these 32 categories included the National Stock Number (a hierarchical pattern used by the US government for requisitions), military ranks for the US Marines, and the standard resource identifier codes stipulated by the government's Energy Analysis and Diagnostic Centers. Although these tail categories each cover few columns, they are national standards and could warrant tope implementations in some contexts.

We implemented topes for each of the 32 most common categories in the spreadsheet data. This yielded 11 topes with enumeration patterns, 10 with list-of-word patterns, 7 with hierarchical patterns, and 4 with a mixture of hierarchical patterns for some formats and list-of-word patterns for other formats. None of these required numeric patterns, perhaps because we biased our extract toward string data, but we noted a few numeric categories in the tail.

We used an early version of the TDE and added primitives to the TDE as needed to implement the topes. We have evaluated the expressiveness of the system in a separate report [31].

5.3 Identifying Questionable Inputs

To evaluate if identifying questionable inputs improved validation accuracy, we considered five conditions.

Condition 1—Current spreadsheet practice: Spreadsheets allow any input. Since no invalid inputs are identified, recall and $F_1 = 0$.

Condition 2—Current web application practice: As Section 2 discussed, web application programmers commonly omit validation for text fields, except when it is convenient to find or create a regexp for the field's data category. For certain enumerable categories, programmers sometimes require users to select inputs from a dropdown or radio button widget.

To simulate current practice, we searched the web for regexps, dropdown widgets, and radio button widgets that could be used to validate the 32 test categories. After searching for hours (more time than a programmer is likely to spend), we found 36 regexps covering 3 of the 32 categories (email, URL, and phone). In addition, based on the visible and internal values in dropdown and radio button widgets that we found on the web, we constructed 34 regexps covering 3 additional categories (state, country, and gender). Finally, under the assumption that programmers would use a checkbox for Boolean values, we constructed a regexp covering the Boolean data category.

In short, our search yielded 71 regexps covering 7 of the 32 categories. While searching the web, we found that approximately half of the sites omitted validation *even for some of these 7 categories*. However, when we calculated F_1 (below), we assumed that programmers would validate these 7 categories and only accept inputs from the remaining 25 without validation.

Condition 3A—Tope rejecting questionable inputs: In this condition, the tope implementations (discussed in Section 5.2) tested each input s, accepting when $\phi(s) = 1$ and rejecting when $\phi(s) < 1$, thus making no use of topes' ability to identify questionable values. For comparability with Condition 2, we restricted topes to a single format each and report on the most accurate below.

Condition 3B—Tope accepting questionable inputs: In this condition, a single-format tope validation function tested each input s, accepting when $\phi(s) > 0$ and rejecting when $\phi(s) = 0$.

Condition 4—Tope warning on questionable inputs: In this condition, a single-format tope validation function tested each input s, accepting when $\phi(s) = 1$ and rejecting when $\phi(s) = 0$. If s was questionable, we simulated the process of asking a human to double-check s, as discussed in Section 4.4, meaning that s was accepted if it was truly valid or rejected if it was truly invalid. The advantage of Condition 4 is that a tope implementation relies on human judgment in difficult cases when $0 < \phi(s) < 1$, thereby raising accuracy. The disadvantage is that a user would need to manually double-check each questionable input. Consequently, we evaluate the tradeoff between increasing accuracy and burdening the user.

As shown in Table 1, validating the 32 categories with single-format topes was more accurate than current practice.

Table 1. Asking the user for help with validating questionable inputs leads to higher accuracy in single-format validation

Condition	F_1
1 – Current spreadsheet practice	0.00
2 – Current web application practice	0.17
3A – Tope rejecting questionable values	0.32
3B – Tope accepting questionable values	0.32
4 – Tope warning on questionable values	0.36

Condition 2, current web application practice, was inaccurate partly because it accepted so many categories of data without validation. F_1 would likely be higher if programmers were in the habit of validating more fields. However, even in the 7 categories where programmers have published regexps on the web, or where we could convert dropdown or radio button widgets to regexps, F_1 was only 0.31 (the same accuracy as Condition 4 in those categories), owing to a lack of regexps for unusual international formats that were present in the EUSES spreadsheet corpus. To achieve higher accuracy than we did with topes, programmers would need to combine numerous international formats into a single regexp for each data category, which stands in stark contrast to current practice.

Condition 4 improved accuracy to 0.36 versus 0.32 in conditions 3A and 3B by identifying questionable inputs and asking a human to double-check certain inputs. This 12% relative difference only required asking the user for help on 4.1% of inputs. Rejecting more inputs would reduce the burden on the user, at the cost of accuracy. For example, rejecting inputs that violate 2 or more production constraints, thus only asking about inputs that violate 1 constraint, would require asking for help on 1.7% of inputs but would reduce F_1 to 0.33. Rejecting inputs that violate any constraint would eliminate the need to ask about inputs, reducing to condition 3A, with F_1=0.32.

5.4 Matching Multiple Formats

To evaluate the benefit of implementing multiple formats, we repeated the analysis of Section 5.3 using more than one regexp or format per category. In particular, we varied the number of regexps or formats (N) from 1 to 5. For example, with N=4, Condition 2 accepted each input s if s matched any of 4 regexps (selected from the 71 regexps). For each N, we report the performance of the best combination of N regexps.

In Conditions 3A, 3B and 4, we calculated $\phi(s) = \max(\mathtt{isa}_i(s))$, where i ranged from 1 through N. For each condition and each value of N, we report the performance of the best combination of N formats.

Figure 3 shows that as N increased, so did accuracy. Different regexps on the web largely made the same mistakes as one another (such as omitting uncommon phone formats), so adding more than 3 regexps did not improve accuracy further. In contrast, increasing the number of formats continued to increase the tope accuracy, since the primitives supported by the TDE made it convenient to implement formats that are difficult to describe as regexps.

For each value of N, a 10% relative difference generally remained apparent between Conditions 3A/B and Condition 4, highlighting the benefit of identifying questionable inputs. In fact, as the number of formats increased, the number of questionable inputs decreased (as adding formats recognized more inputs as valid). By N=5, the 12% relative difference between Condition 4 and Conditions 3A/B required asking the user for help on only 3.1% of inputs.

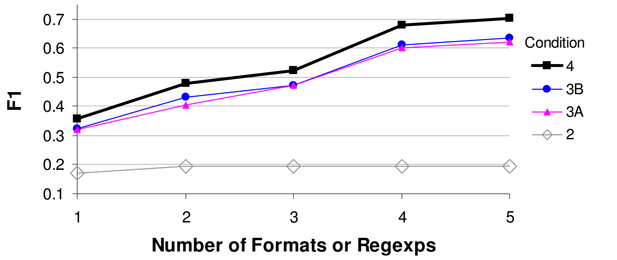


Figure 3. Including additional formats improves accuracy

5.5 Reusability

To evaluate reusability, we tested if the topes that we implemented for spreadsheet data could accurately validate web application data.

Of the 32 topes implemented, 7 corresponded to fields in Google Base. We validated an eighth category in Google Base, mailing addresses, by concatenating existing address line, city, state, and country topes with separators and a zip code field. (That is, these topes corresponded to distinct cells in spreadsheets but were concatenated into one field in Google Base.) In addition to these 8 Google Base fields, we reused topes to validate the 5 Hurricane Katrina text fields.

We manually validated each of the 1300 test values and computed the topes' F_1, as shown in Figure 4. Comparing these results to Figure 3 reveals that the topes actually were *more* accurate on the web application data than on the spreadsheet data, largely because the spreadsheet corpus demonstrated a wider variety of patterns for each category. In particular, for Hurricane Katrina data, accuracy was so close to 1 even with a single format that including more formats yielded little additional benefit.

For each of the 32 topes, we also informally assessed what factors may limit tope reusability. Language may affect list-of-words patterns that rely on word length and punctuation. Location may limit reusability of formats used only within certain geographical boundaries, such as phone number formats. Organization boundaries may affect topes describing organization-specific data, such as academic course titles. Finally, industry concerns may limit reusability of topes, such as manufacturer product codes.

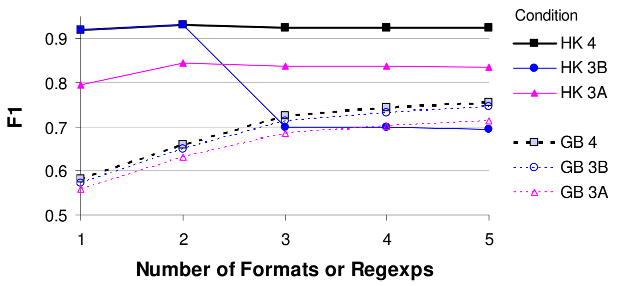


Figure 4. Accuracy remained high when reusing topes on Hurricane Katrina (HK) and Google Base (GB) data.

5.6 Data Transformation Results

In order to evaluate if using topes to put inputs into a consistent format would have helped Hurricane Katrina aggregators identify duplicate values, we implemented transformations for each 5-format tope from the less commonly used formats to the most commonly used, which we specified as the preferred format for each tope. We randomly extracted 10000 values for each of the 5 categories and transformed data into the preferred format.

We found approximately 8% more duplicates after transformation, including 16% more first name duplicates, 9% more last name duplicates, 9% more address line duplicates, 8% more phone number duplicates, and 2% more city name duplicates. We discovered so few additional city duplicates because almost all inputs contained "New Orleans", leaving few new duplicates to discover (largely by changing uppercase names to title case). We suspect that topes would be even more effective on a more diverse city dataset.

6. RELATED WORK

This paper introduced a new kind of abstraction that describes how to recognize and transform values in a category of data, leading to a new data validation technique. This section compares topes to existing data category abstractions.

Type systems: "A type system is a tractable syntactic method for proving the absence of certain behaviors by classifying phrases according to the kinds of values they compute" [20]. Types enable a compiler to identify code that attempts to call invalid operations. Several programming platforms associate formal types with web forms' text fields. Depending on the language paradigm, these systems assign inputs to an instance of an object-oriented class [4], a functional programming monad or a free variable [13], or an instance of a generated specialization of a generic type [21].

Whereas topes can identify questionable inputs, membership in a formal type is a binary question—either an expression is or is not an instance of a type. Binary membership in a type makes it possible to prevent certain undesirable program behaviors at compile time, but the absence of fuzziness limits types' usefulness for validating ambiguous data categories at runtime. In contrast, each `isa` function serves as a membership function for a fuzzy set [36], which allows elements to have a degree of membership. This enables topes to identify questionable inputs for double-checking.

Grammars: PowerForms [5] and phpClick [26] allow programmers to specify regexps for validating web form fields. Lapis patterns [17] and Apple data detectors [18] recognize values embedded in text by matching data to grammars that exceed regexps in expressive power.

PowerForms is particularly interesting, in that as a user enters an input s, PowerForms annotates the field with a green icon (if s matches the regexp), a yellow icon (if s does not yet match but could match if the user types more characters), or a red icon (if s is not a prefix of the regexp). This bears some resemblance to our technique, in identifying "yellow" inputs that are not valid but remain promising. However, PowerForms still ultimately requires a regexp, which is difficult or impossible to construct for many categories.

Existing validation techniques based on regexps, CFGs, and other grammars still leave inputs in multiple formats at the end of validation, whereas validating with topes transforms data into the format needed by the main application.

Machine learning for information extraction: Many machine learning algorithms train a model to notice certain features of values. Information extraction systems apply the model to identify new instances embedded in natural language [16]. These algorithms take advantage of contextual features. For example, the words "worked for" may precede a company name in natural language. However, for the data validation problem that concerns programmers, these contextual cues are not available or not useful. For example, if a programmer places a "Phone:" label beside a form field (so users know what data belongs there), then that label provides no information about whether a particular input is valid.

Although these limitations apparently prevent using these algorithms for input validation, the information extraction community's notion of a "named entity" has strongly influenced our conception of topes, as described in Sections 3 and 4.

Platform-specific numeric and formula constraints: Cues infers numeric constraints over web service data [25], and Forms/3 infers numeric constraints over spreadsheet cells [6].

Several systems use units and dimensions to infer constraints over how formulas can *combine* spreadsheet cells. For example, the σ-calculus associates types with cells (based on the placement of labels at the top of columns and at the left end of rows), and the calculus specifies how types propagate through the spreadsheet. If two cells with different types are combined, then their type is generalized if an applicable type exists (e.g.: "3 apples + 3 oranges = 6 fruit"), or else an error message is shown [9]. Slate does similar validation using physical units and dimensions, such as kilograms and meters [8]. Similar unit-based type systems, with minimal support for constraint inference, are supported in other programming environments (such as Java [2] and ML [15]).

These approaches only apply to numeric data in particular platforms, such as web services, spreadsheets, and Java. In contrast, topes can validate strings (including strings representing numbers), and they are platform-agnostic.

Data cleaning techniques: Topes help to ensure data quality by identifying questionable data and transforming data to a consistent format. In current practice, database administrators or other skilled professionals perform these data cleaning tasks offline.

Researchers have provided tools to simplify data cleaning. For example, the Potter's Wheel is an interactive tool that presents database tables in a spreadsheet-like user interface, allowing administrators to edit specific cells manually to correct errors or to put data into a consistent format [24]. From these edits, the Potter's Wheel infers transformation functions, which administrators can save, reload, and replay on other data values that require the same cleaning steps. See [23] for a survey of related tools.

Topes include similar functions for recognizing and transforming data. While topes may prove useful for helping administrators to clean data in databases, we have focused in this paper on helping programmers write applications that use topes to clean data at runtime. As a result, web applications and other programs can work with users to clean inputs *before* the data reach the database, thereby reducing offline work for database administrators.

Microformats: Microformats are a labeling scheme where a web developer affixes a "class" attribute to an HTML element to specify a category for the element's text. For example, the web developer might label an element with "tel" if it contains a phone number. This does not actually validate the text. It simply labels the element with a string that programmers commonly recognize as meaning "phone number". Commonly recognized labels are published on a wiki (http://microformats.org/wiki/). Other people can write programs that download a labeled web page and retrieve phone numbers from the HTML. The values would then require validation and, perhaps, transformation to a common format—the problems addressed by topes.

We are working to integrate microformats and topes [29], as people could upload tope implementations for common labels to the microformat wiki. Then, when other programmers need to download data with a certain label, they could find relevant tope implementations on the wiki and use them to validate and transform data.

7. CONCLUSION AND FUTURE WORK

This paper presented a new technique that improves the accuracy and reusability of validation code. The technique is supported by a new kind of data abstraction, which describes how to recognize and transform instances of a data category.

In our evaluation, we noted that categories often repeat across a range of spreadsheets and web sites. To support reuse, we are extending the TDE with a repository system where people can publish and find tope implementations. Repository search mechanisms will enable software engineers to identify suitable tope implementations based on quality criteria and based on relevance to new applications.

Our evaluation's main limitation is that it was an experiment on a data extract, rather than an "in vivo" evaluation of the TDE in an actual software engineering setting. Developing a repository will enable us to collect actual tope implementations as well as feedback from people using topes in real applications. This will facilitate incremental TDE improvements to further assist software engineers as they implement and reuse topes to validate data.

8. ACKNOWLEDGMENTS

This work was funded in part by the EUSES Consortium via NSF (ITR-0325273) and by NSF under Grants CCF-0438929 and CCF-0613823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] Adler, R., Feldman, R., and Taqqu, M. *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, Birkhäuser Verlag, 1998.

[2] Allen, E. et. al. *The Fortress Language Specification*, Sun Microsystems, 2006.

[3] Aslam, T., Krsul, I., and Spafford, E. *Use of a Taxonomy of Security Faults*. Tech. Rpt. TR-96-051, Purdue University, 1996.

[4] Bhasin, H. *Asp.NET Professional Projects.* Muska & Lipman Publishers, 2002.

[5] Brabrand, C., Møller, M., Ricky, M., and Schwartzbach, M. PowerForms: Declarative Client-Side Form Field Validation. *World Wide Web*, *3*, 4 (Dec. 2000), 205-214.

[6] Burnett, M., et al. End-User Software Engineering with Assertions in the Spreadsheet Paradigm. *Proc. Intl. Conf. Soft. Eng.*, 2003, 93-103.

[7] Chakrabarti, S. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, 2002.

[8] Coblenz, M., Ko, A., and Myers, B. Using Objects of Measurement to Detect Spreadsheet Errors. *Proc. Symp. Visual Lang. and Human-Centric Computing*, 2005, 314-316.

[9] Erwig, M., and Burnett, M. Adding Apples and Oranges. *Proc. 4th Intl. Symp. Practical Aspects of Declarative Lang.*, 2002, 173-191.

[10] Fisher II, M., and Rothermel, G. *The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms*. Tech. Rpt. 04-12-03, University of Nebraska—Lincoln, 2004.

[11] Flanagan, D. JavaScript: The Definitive Guide. O'Reilly, 2006.

[12] Hall, J. A Risk and Control-Oriented Study of the Practices of Spreadsheet Application Developers. *Proc. 29th Hawaii Intl. Conf. System Sciences*, 1996, 364-373.

[13] Hanus, M. Type-Oriented Construction of Web User Interfaces. *Proc. 8th SIGPLAN Symp. Principles and Practice of Declarative Programming*, 2006, 27-38.

[14] Howard, M., LeBlanc, D., and Viega, J. *19 Deadly Sins of Software Security.* McGraw-Hill, 2005.

[15] Kennedy, A. *Programming Languages and Dimensions*. PhD thesis, Tech. Rpt. 391, University of Cambridge, 1996.

[16] Marsh, E., and Perzanowski, D. MUC-7 Evaluation of IE Technology: Overview of Results. *7th Message Understanding Conf.*, 2001.

[17] Miller, R., and Myers, B. Lightweight Structured Text Processing. *Proc. 1999 USENIX Annual Technical Conf*, 1999, 131-144.

[18] Nardi, B., Miller, J., and Wright, D. Collaborative, Programmable Intelligent Agents. *Comm. ACM*, *41*, 3 (Mar. 1998), 96-104.

[19] Panko, R. What We Know About Spreadsheet Errors. *J. End User Computing*, *10*, 2 (Spring 1998), 15-21.

[20] Pierce, B. *Types and Programming Languages*, MIT Press, 2002.

[21] Plasmeijer, R., and Achten, P. *The Implementation of iData—A Case Study in Generic Programming*. Tech Rpt. TCD-CS-2005-60, Dublin University, 2005.

[22] Porter, M. An Algorithm for Suffix Stripping. *Program*, *14*, 3 (July 1980), 130-137.

[23] Rahm, E., and Do, H. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bulletin*, *23*, 4 (Dec. 2000), 3-13.

[24] Raman, V., and Hellerstein, J. Potter's Wheel: An Interactive Data Cleaning System. *Proc. 27th Intl. Conf. Very Large Data Bases*, 2001, 381-390.

[25] Raz, O., Koopman, P., and Shaw, M. Semantic Anomaly Detection in Online Data Sources. *Proc. 24th Intl. Conf. Software Engineering*, 2002, 302-312.

[26] Rode, J. *Web Application Development by Nonprogrammers: User-Centered Design of an End-User Web Development Tool*. PhD Thesis, Virginia Polytechnic Institute, 2005.

[27] Rothermel, G., et al. WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation. *Proc. Intl. Conf. Soft. Eng.,* 2000, 230-239.

[28] Scaffidi, C. Unsupervised Inference of Data Formats in Human-Readable Notation. *Proc. 9th Intl. Conf. Enterprise Integration Systems – HCI Volume*, 2007, 236-241.

[29] Scaffidi, C., Shaw, M. Accommodating Data Heterogeneity in ULS Systems. *2nd Intl. Workshop on Ultra-Large-Scale Software-Intensive Systems*, at the 30th Intl. Conf. Software Engineering, to appear.

[30] Scaffidi, C., Myers, B., and Shaw, M. Challenges, Motivations, and Success Factors in the Creation of Hurricane Katrina "Person Locator" Web Sites. *Psychology of Programming Interest Group Workshop*, 2006.

[31] Scaffidi, C., Myers, B., and Shaw, M. *The Topes Format Editor and Parser*. Tech Rpt. CMU-ISRI-07-104, Carnegie Mellon University, 2007.

[32] Scaffidi, C., Myers, B., and Shaw, M. Toped: Enabling End-User Programmers to Describe Data, *Conf. on Human Factors in Computing Systems – Work-in-Progress posters*, 2008, to appear.

[33] Scaffidi, C., Shaw, M., and Myers, B. Estimating the Numbers of End Users and End User Programmers. *Proc. 2005 Symp. Visual Lang. and Human-Centric Computing,* 2005, 207-214.

[34] Scaffidi, C., Shaw, M., and Myers, B. Games Programs Play: Obstacles to Data Reuse, *2nd Workshop on End User Soft. Eng*, 2006

[35] Shaw, M. Larger Scale Systems Require Higher-Level Abstractions. *5th Intl. Workshop on Soft. Spec. and Design*, 1989, 143-146.

[36] Zadeh, L. *Fuzzy Logic*. Tech Rpt. CSLI-88-116, Stanford University, 1988.