Spanner Workbench
Design Document

January 1, 2024

The *Spanner Workbench* is intended to become of an implementation that allows declarative expression of Information Extraction tasks. Spanner Workbench is based on the (rule-based) approach to Information Extraction as expressed by XLog [**?**], SystemT [**?**], and DeepDive [**?**]. In this approach, complex extractor logic is expressed by combining *primitive extractors* (e.g., tokenizers, dictionaries, part-of-speech taggers, regular expressions matchers, ...) through algebraic combinators.

In particular, Spanner Workbench implements the theoretical framework of Document Spanners [**?**], which formalizes the rule-based approach to Information Extraction cited above. In this framework, a *document spanner* (or just spanner for short) is any function that takes a document as input and yields a relation over the *spans* (text intervals) of this input. Various languages for expressing document spanners have recently been studied. Spanner Workbench will implement spanner$_{w}orkbench$ [**?**], $whichistheclosureofregularexpressionformulasunderrecursiveData$

This design document is intended to describe the concrete syntax of IE programs in version 0.1 of the Spanner Workbench.

1 Syntax and Semantics

Primitive extractors. The only primitive extractors that we intend to support in version 0.1 are *functional regex formulas*. Essentially, a regex formula is a regular expression with capture variables. In the literature [**?**] such functional regex formulas are often given a very simple syntax, e.g.,

$$\gamma ::= \emptyset \mid \varepsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\},$$

where ε represents the empty string, σ ranges over alphabet letters, and x ranges over variable names. While this minimalistic syntax is appealing for theoretical investigation, it is cumbersome to use in practice. Indeed, regular expressions in practice have many convenient abbreviations (see e.g., `https://www.pcre.org/current/doc/html/pcre2pattern.html`).

As a concrete syntax for regex formulas, it is proposed to use the syntax of Perl regular expressions, where we interpret *named capture groups* to indicate variable capture. (See `https://www.regular-expressions.info/named.html` for a discussion on named capture groups.) Unnamed capture groups are not supported ignored. Backreferences are not allowed (since this allows recognition of non-regular languages). As usual, functional regex formulas are given a *all-match* semantics instead of the unique match semantics supported by Python/Perl/POSIX regular expressions. We enforce functionality by syntactic checks.

While in principle Perl regular expression allow naming capture groups using both the .NET syntax and the python syntax, we choose the .NET syntax.

Some examples:

- The expression `a(?<x>b)c(?<y>d)` has two named capture groups: `(?<x>b)` and `(?<y>d)` where the former matches b (binding the matched span to x, and the latter matches d (binding the matched span to y). As such, matching the entire expression against the document $abcd$ is successful, and outputs the singleton relation with the tuple t where $t(x) = [1, 2 >$ and $t(y) = [3, 4 >$.
- The expression `[.*(?<year>\d\d\d\d).*(?<amount>\d+)\sEUR` matches the input document `In 2019 we earned 2000 EUR`, extracting 2019 in `year`, and 2000 in `amount`
- The above examples use the .NET syntax. The Python syntax would be `a(?P<x>b)c(?P<y>d)` and `[.*(?P<year>\d\d\d\d).*(?P<amount>\d+)\sEUR`, respecitvely.

Other important decisions:

- We implicitly assume that a regular expression formula is prefixed by $.*$ and suffixed by $.*$ (which means it will find any occurrence of the regex in the string), *unless* it starts with the anchors (`^` and `$`) that ask it to match the entire string? This means that in the second example above, there was no need explicitly add `.*` at the beginning to make sure that matching could start anywhere.
- We assume that the input string is UTF8 encoded. This is compatible with ASCII, but also allows to process international text.

Rules, Programs and Queries In version 0.1, a *program* will essentially be a (standard) Datalog program where:

- all predicate symbols are *intensional*
- regex formulas function as *extensional* relations
- all variables are of *span* type. (I.e., every variable can only store spans, not strings or other data).

This corresponds to what is described in "Recursive Programs for Document Spanners", published in ICDT 2019.

Syntactic considerations:

- It is proposed to use <- for separating rule heads from rule bodies.
- It is proposed to not impose any syntactic restrictions on relation and variable names. (In some Datalog variants, variable names must start with a capital ...).
- Inside a rule body, a regex formulas is denoted as follows: RGX<f>$(x_1, \ldots, x_n)$. Here, f is a regex formula and $x_1, \ldots, x_n$ is a listing of all the variables in f. (Note that these variables may appear in a different order in f).

- To specify that a regex formula inside a rule needs to executed on a specific input document, we use epxressions like

  ```
  extract RGX<f>(x_1,\dots,x_n)  from  d
  ```

  that specifies the regex formula as well as where to extract from. Here, d is allowed to be (1) a constant string or (2) a string variable. We allow defining and instantiating string variables by the syntax

  $$d = \text{'some string'},$$

  (which initializes d to the corresponding string literal, or

  $$d = \text{read('data/a.txt')}$$

  which loads the contents of the file `data/a.txt` into string variable d.

Left for future work:

- Provisions for modularity (e.g., grouping rules in modules, having namespaces to avoid nameclashes etc) will not be included in version 0.1, and are postponed to a later version.
- Syntactic sugar for things that are expressible, but cumbersome to expres (e.g., the ς operator, same length, prefix, ...) will not be included in version 0.1, and are postponed to a later version.

References

[1] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. Systemt: An algebraic approach to declarative information extraction. In *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden*, pages 128–137, 2010.

[2] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51, 2015.

[3] L. Peterfreund, B. ten Cate, R. Fagin, and B. Kimelfeld. Recursive programs for document spanners. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, pages 13:1–13:18, 2019.

[4] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1033–1044, 2007.

[5] C. Zhang, C. Ré, M. J. Cafarella, J. Shin, F. Wang, and S. Wu. Deepdive: declarative knowledge base construction. *Commun. ACM*, 60(5):93–102, 2017.