Spanner Workbench
Design Document

October 28, 2019

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 RGXLog [**?**], which is the closure of regular expression formulas under recursive Datalog programs.

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 Python 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 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.

Some examples:

- The expression `a(?P<x>b)c(?P<y>d)` has two named capture groups: `(?P<x>b)` and `(?P<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 `[.*(?P<year>\d\d\d\d).*(?P<amount>\d+)\sEUR` matches the input document `In 2019 we earned 2000 EUR`, extracting 2019 in `year`, and 2000 in `amount`

Issues to resolve:

- Do we require that a regex formula always matches the entire document? Or do we implicitly assume that it is prefixed by $.*$ and suffixed by $.*$ (i.e., find any occurrence), unless it starts with the anchors (`^` and `$`)? In the second example above, we explicitly added `.*` at the beginning to make sure that matching could start anywhere.
- The .NET framework allows named groups by means of the slightly simpler syntax `(?<name>e)` where name is the variable name and e is the regex. I would actually prefer to use this syntax. (Note that Java and Perl allow both).
- What is the input alphabet that we are considering? It may be simplest to assume that all input 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 ...).

Issues to resolve:

- How do we specify that a program needs to be run on a given input document ? One possibility would be to have some kind of *query expression* like

```
extract R(x,z,_) from '/home/stijn/test.txt'
```

that specifies the target relation to extract, as well as the file. It is only when we get such a query expression that the program is evaluated.

- Do we already provision for modularity (e.g., grouping rules in modules, having namespaces to avoid nameclashes etc) in this version? We can probably omit it for now.
- Do we foresee already in version 0.1 syntactic sugar for things that are expressible, but cumbersome to express? E.g., the ς operator, same length, prefix, ...

2 Grammar

TO BE DONE