Equivalence between models for regex-based information extraction with errors

25 de agosto de 2025

Resumen

We formalize and prove the equivalence between two models for regex-based information extraction with errors: (1) a semantics operating on the original document with error annotations, and (2) a semantics working on a modified document with edit operations. We demonstrate a bijective mapping between matches in both models, preserving spans and error operations under consistent index transformations. This results enable interoperability between systems using either approach.

1. Preliminaries

1.1. Basic definitions

Document and spans. A document d is a string over a finit alphabet Σ, we write $d = a_0a_2...a_{n-1}$ to denote a document of legth $|d| = n$. A span over a document d is a pair $s = [i, j\rangle$ of natural numbers i and j with $0 \leq i \leq j \leq |d|$. s is associated with a substring of d from position i to position $j - 1$. We denote this substring by $d(s)$. Given two spans $s_1 = [i_1, j_1\rangle$ and $s_2 = [i_2, j_2\rangle$, the concatenation of s_1 and s_2 is defined as $s_1 \cdot s_2 = [i_1, j_2\rangle$, whenever $j_1 = i_2$.

Mappings. Let V be a fixed set of variables, disjoint from Σ. A *mapping* is a function μ from a finite set of variables $dom(\mu) \subseteq V$ to spans. The empty mapping, denoted by $\varnothing$, is the only mapping such that $dom(\varnothing) = \varnothing$. Similarly, $[x \to s]$ denotes the mapping whose domain only cointains the variable x, which it assign to span s.

Document spanners. A document spanner is a function that map every document d to a set of mappings M, such that the range of each $\mu \in M$ are spans of d modeling the process of extracting information from d. We will work with 3 models for defining spanners: RGX, ref-words and automatas.

Regex Formulas. A *regex-formula* (**RGX** for short) is a regular expression extended with variables (called capture variables). Formally, we define the syntax with the recursive rule:

$$e = \varnothing \mid \epsilon \mid e \mid e \vee e \mid e \cdot e \mid e^* \mid x\{\ e\ \}$$

Variable-set automata. A *variable-set automaton*(VA) can be understood as an $\epsilon - NFA$ that is extended with capture variables in a way analogous to RGX; that is, it behaves a usual finite state automaton with $\epsilon-$transitions, except it can also open and close variables. Formally, a VA $\mathcal{A}$ is tuple $(Q, \Sigma, V, q_0, F, \delta)$, where Σ is a finite set of alphabet symbols, V is a finite set of variables, Q is a finite set of states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states and δ is a transition relation consisting of letter transitions of the form (q, a, q') and variable transition of the form $(q, x \vdash, q')$ or $(q, \dashv x, q')$, where $q, q' \in Q$, $a \in Q$ and $x \in V$. We define the set $Vars(\mathcal{A})$ as the set of all variables that occur in $\mathcal{A}$

A configuration of a VA $\mathcal{A}$ over a document $d = a_0a_1...a_{n-1}$ is a tuple (q, i), where $q \in Q$ is the current state and $i \leq n$ is the current position in the document. A run ρ of $\mathcal{A}$ is sequence of the form:

$$\rho := (q_0, i_o) \xrightarrow{a_0} (q_1, i_1) \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} (q_{n-1}, i_{n-1})$$

where $a_i \in \Sigma \cup \{x \vdash, \dashv x \mid x \in V\}$ and (q_i, a_{i+1}, q_{i+1}). Moreover, $i_0...i_{n-1}$ is a non-decreasing sequence such that $i_0 = 1$ and $i_{n-1} = |d|$, and $i_{j+1} = i_j + 1$ if $a_{j+1} \in \Sigma$ and $i_{n-1} = i_j$ otherwise.

Ref-words. For a finite set V of variables, ref-words are defined over the extended alphabet $\Sigma \cup \Gamma_V$, where $\Gamma_v := \{\vdash x, \dashv x \in V\}$, we assume that Γ_V is disjoint with Σ. Ref-words extend strings over Σ by encoding opening $(x \vdash)$ and closing $(\dashv x)$ of variables. A ref-words $r \in (\Sigma \cup \Gamma_V)^*$ is valid if every ocurring variable is opened and closed exactly once. For every valid ref-word r over $(\Sigma \cup \Gamma_V)$ we define $Vars(r)$ as the set of variables $x \in V$ which ocurr in the ref-word. More formally

$$Vars(r) := \{x \in V \mid \exists r_x^{prev}, r_x, r_x^{post} \in (\Sigma \cup \Gamma_V)^*.\ r = r_x^{prev} \cdot x \vdash \cdot r_x \cdot \dashv x \cdot r_x^{post}\}$$

Intuitively, each valid ref-word encondes a mapping for some document d, the variable markers encode where the spans begin and end. Formally, we define functions doc and tup that, given a valid ref-word, output the corresponding document and mapping. The morphism $doc : (\Sigma \cup \Gamma_V)^* \to \Sigma^*$ is defined as:

$$\text{doc}(a) := \begin{cases} a & \text{if } a \in \Sigma \\ \varepsilon & \text{if } a \in \Gamma_V \end{cases}$$

Now, we have that every valid ref-word has a unique factorization

$$r = r_x^{prev} \cdot x \vdash \cdot r_x \cdot \dashv x \cdot r_x^{post}$$

, for each $x \in Vars(r)$. We can define de function tup as

$$tup(r) := \{x \to [i_x, j_x\rangle \mid x \in Vars(r), i_x = |doc(r_x^{prev})|, j_x = i_x + |doc(r_x)|\}$$

The usage of the doc morphism ensures that indices i_x and j_x refer to positions in the document and do not consider other variable operations.

1.2. REmatch Semantics

REmatch's formal semantics are defined through inductive rules that specify how patterns match documents and produce variable bindings. The core semantic definition appears in Table 1, which we explain below.

1.2.1. Semantic Foundations

The semantics operates on two levels:

- $[\![e]\!]_d$: The set of all $(span, mapping)$ pairs where e matches document substring $d[span]$
- $[\![e]\!]_d$: The final variable mappings produced by matching e against entire document d

Cuadro 1: Inductive semantics of REQL queries

Pattern	**Semantics**
a	$[\![a]\!]_d = \{(s, \emptyset) \mid s \in \text{span}(d) \wedge d(s) = a\}$
.	$[\![.]\!]_d = \{([i, i+1\rangle, \emptyset) \mid 0 \leq i < \lvert d\rvert\}$
$[w]$	$[\![[w]]\!]_d = \{(s, \emptyset) \mid d(s) \in \text{set}(w)\}$
`!x{`e`}`	$[\![\texttt{!x\{}e\texttt{\}}]\!]_d = \{(s, \mu) \mid \exists (s, \mu') \in [\![e]\!]_d : \mu = \mu' \cup [x \mapsto s]\}$
$e_1 e_2$	$[\![e_1 e_2]\!]_d = \{(s_1 \cdot s_2, \mu_1 \cup \mu_2) \mid (s_i, \mu_i) \in [\![e_i]\!]_d\}$
$e_1 \lvert e_2$	$[\![e_1 \lvert e_2]\!]_d = [\![e_1]\!]_d \cup [\![e_2]\!]_d$
e^*	$[\![e^*]\!]_d = \bigcup_{k \geq 0} [\![e^k]\!]_d$

1.2.2. Key Semantic Rules

The table shows how REmatch composes matches from subexpressions: This semantic model enables REmatch to:

- Track all possible matches simultaneously
- Maintain correct variable binding scopes
- Compose matches algebraically via span concatenation

1.3. Edit Distance and Error Formalization

1.3.1. The Edit Distance Problem

In document processing with regular expressions, we often encounter texts with imperfections such as:

- Typos (e.g., documment instead of document)
- Missing characters (e.g., docment)

- Extra insertions (e.g., docuument)

  .[a]¿.[a]bc". ins(1, b), ins(c,2) — ins (1,c) ins(1,b)

The *edit distance* between two strings D and D', denoted by $\Delta(D, D')$, is the minimum number of elementary edit operations required to transform D into D'. We consider three fundamental operations:

Given a document $D = a_1 \cdots a_n$:

- **Insertion**: $\mathsf{ins}(p, c)$ - Insert character c at position p

$$D' = a_1 \cdots a_p c a_{p+1} \cdots a_n$$

- **Deletion**: $\mathsf{del}(p)$ - Delete character at position p

$$D' = a_1 \cdots a_{p-1} a_{p+1} \cdots a_n$$

- **Substitution**: $\mathsf{sub}(p, c)$ - Replace a_p with c

$$D' = a_1 \cdots a_{p-1} c a_{p+1} \cdots a_n$$

2. Edit Operation Application

2.1. Formal Definition

Given a document $D \in \Sigma^*$ (where $D = d_1 d_2 \cdots d_n$) and a sequence of edit operations $E = [op_1, \ldots, op_m]$, we define the transformed document $\mathsf{apply}(D, E)$ recursively as:

$$\mathsf{apply}(D, E) = \begin{cases} D & \text{if } E = \emptyset \\ \mathsf{apply}(\mathsf{op}_1(D), [op_2, \ldots, op_m]) & \text{otherwise} \end{cases}$$

where each atomic operation op_i modifies the document as follows:

$$\begin{aligned} \mathsf{ins}(p, c)(D) &= d_1 \cdots d_p c d_{p+1} \cdots d_n \quad (0 \le p \le n) \\ \mathsf{del}(p)(D) &= d_1 \cdots d_{p-1} d_{p+1} \cdots d_n \quad (1 \le p \le n) \\ \mathsf{sub}(p, c)(D) &= d_1 \cdots d_{p-1} c d_{p+1} \cdots d_n \quad (1 \le p \le n) \end{aligned}$$

2.2. Properties

For any document D and valid operation sequence E:

1. **Termination**: $\mathsf{apply}(D, E)$ always terminates in $O(|E|)$ steps
2. **Length Change**: $|\mathsf{apply}(D, E)| = |D| + \#\mathsf{ins} - \#\mathsf{del}$
3. **Order Sensitivity**: $\mathsf{apply}(D, [op_1, op_2]) \neq \mathsf{apply}(D, [op_2, op_1])$ in general

2.2.1. Example

Let $D = \texttt{algorithm}$ and $E = [\mathsf{ins}(3, \texttt{h}), \mathsf{del}(6), \mathsf{sub}(2, \texttt{L})]$. Then:

$$\begin{aligned}\mathsf{apply}(D, E) &= \mathsf{apply}(\mathsf{ins}(3, \texttt{h})(\texttt{algorithm}), [\mathsf{del}(6), \mathsf{sub}(2, \texttt{L})]) \\ &= \mathsf{apply}(\texttt{alghorithm}, [\mathsf{del}(6), \mathsf{sub}(2, \texttt{L})]) \\ &= \mathsf{apply}(\texttt{alghorthm}, [\mathsf{sub}(2, \texttt{L})]) \\ &= \texttt{alLgrithm}\end{aligned}$$

3. Models for information extraction with errors

Now, we will consider two diferent approaches to aboard the edit distance problem in Rematch engine.

3.1. Error over the original document

Normally, when we think in some computer system that correct typos, we imagine that this system say us: "This is your document, you have to make this changes to correct it"

In this approach, we will work on that idea. Given a document $d \in \Sigma^*$ (where $d = d_1 d_2 \cdots d_n$) and a REQL expression R, we define the semantics of R with k error as it follows:

$$[\![R]\!]_d^{\leq K} = \{(s, \mu, E) \mid s \in \mathsf{span}(\mathsf{d}) \wedge \mathsf{apply}(d, E) \in \mathcal{L}(R) \wedge |E| \leq K\}$$

Nota

Santiago: Para este caso, apply falla, ya que en esta semantica, como los errores se hacen sobre los spans del documento original, pueden haber varios errores en la misma posicion, lo que hace que el apply falle. Las opciones para arreglarlo son concatenar errores en una misma posicion o usar un apply que lleve un trackeo de la posicion

Here, we output a span and a mapping over the original document and the list of errors that the user have to make in de document d, to complete de match.

As we work over spans in the original document, we can have errors that apply in the same position of the document, for that reason, es crucial to mantain a correct order in the edits to get te correct match.

3.1.1. Example

Given the document $d = a$ and the regex $e = a!x\{b\}c$, with $k = 2$ two errors available, the following is a match in this sematics.

$$\Big([0,1\rangle,\ \mu := \{x \mapsto [1,1\rangle\},\ E = [\mathsf{ins}(1,\mathsf{b}), \mathsf{ins}(1,\mathsf{c})]\Big)$$

3.1.2. Idea para computar

Usar un automata de levenstein que lleve registro de las posiciones del documento donde se hicieron los errores.

3.2. Error over a modified document

The last approach can be confusing, as we don't use spans of the document with the changes, is not really clear how to make the edits. Moreover, is neither clear how to build the errors sequence if we work over the original document.

The other option is the straightforward version. Take a document d, compute all the possibles documents d' that the edit distance between d and d' is less than k and prove if d' matches de REQL expresion.

This is the semantics for this approach:

Given a document $d \in \Sigma^*$ (where $d = d_1 d_2 \cdots d_n$) and a REQL expression R, we define the semantics of R with k error as it follows:

$$[\![R + k]\!]_d = \{(s, \mu, E) \mid \exists d'.\mathsf{apply}(d, E) = d' \wedge (s, \mu) \in [\![R]\!]_{d'} \wedge |E| \leq K\}$$

We make a match in a new document d' that is the result of apply the edits in E over d. The span and the mapping is over d' because we only compute d' and treat it like the original document.

3.2.1. Example

Using the same example as before, with $d = a$, $e = a!x\{b\}c$ and $k = 2$ the following is a valid match in the new semantics:

$$([0,3\rangle, \mu := [x \mapsto [1,2\rangle], E = (ins(1,b), ins(c,2))$$

The document d' that make the match is $d' = abc$, the span and mapping is over this document.

Now we give the user the new document and the list of edits that we make over the original to get the new one.

3.2.2. Idea para computar

Se puede seguir una idea similar a la que usa en el modulo de filtering, aplicar un algoritmo de light search sobre el documento original que, si es que no se llega a encontrar ningun match, pueda agregar al documento los pasos que faltaron para obtener al menos un match

4. Equivalence Proof

Theorem 1. *For any document d, REQL expression R, and error budget k, there exists a bijection between:*

$$[\![R]\!]_d^{\leq k} \leftrightarrow [\![R+k]\!]_d$$

Proof

First, let's define some functions that will help us with the proof:

Forward and Backwards: Para un documento d, secuencia de errores E, we define $\mathsf{dels}_{\leq p}(E), ins_{\leq p}(E)$ as counters of the deletions and insertions edits that are made before position p

For example, with $E = (subs(2,a), del(3), del(4), ins(c,9), ins(k,10)$ we have $\mathsf{dels}_{\leq 3}(E) = 1, \mathsf{dels}_{\leq 1}(E) = 0, \mathsf{ins}_{\leq 10}(E) = 2$.

Let $\Delta_{\leq p}(E) = \mathsf{ins}_{\leq p}(E) - \mathsf{dels}_{\leq p}(E)$. Using this, we can define the $\mathsf{forward}, \mathsf{backward}$ functions as it follows:

1. **For Spans**:

$$\mathsf{forward}([i,j\rangle, E) = [i, j + \Delta_{\leq j}(E)\rangle$$

$$\mathsf{backward}([i',j'\rangle, E) = [i', j' - \Delta_{\leq j'}(E)\rangle$$

2. **For Mappings**:

$$\mathsf{forward}(\mu, E)(x) = \mathsf{forward}(\mu(x), E)$$

$$\mathsf{backward}(\mu', E)(x) = \mathsf{backward}(\mu'(x), E)$$

3. **For Errors**:
Given $E = [op_1, ..., op_m]$ in d, convert into E' in d':

$$\mathsf{forward\_ops}(E) = [\mathsf{adjust}(op_1), ..., \mathsf{adjust}(op_m)]$$

where for $op = \mathsf{ins}(p,c), \mathsf{del}(p), \mathsf{sub}(p,c)$:

$$\mathsf{adjust}(op_k) = \begin{cases} \mathsf{ins}(p + \Delta_{\leq p}([op_1, ..., op_{k-1}]), c) \\ \mathsf{del}(p + \Delta_{\leq p}([op_1, ..., op_{k-1}])) \\ \mathsf{sub}(p + \Delta_{\leq p}([op_1, ..., op_{k-1}]), c) \end{cases}$$

Now, the inverse process, convert d' into d:

$$\mathsf{backward\_ops}(E') = [\mathsf{adjust}^{-1}(op'_1), \ldots, \mathsf{adjust}^{-1}(op'_m)]$$

where for each operation $op' = \mathsf{ins}(p', c), \mathsf{del}(p'), \mathsf{sub}(p', c)$:

$$\mathsf{adjust}^{-1}(op'_k) = \begin{cases} \mathsf{ins}(p' - \Delta_{\leq p'}([op'_1, ..., op'_{k-1}]), c) \\ \mathsf{del}(p' - \Delta_{\leq p'}([op'_1, ..., op'_{k-1}])) \\ \mathsf{sub}(p' - \Delta_{\leq p'}([op'_1, ..., op'_{k-1}]), c) \end{cases}$$

Finally, we can continue with the proof, now we have to prove both sides of the equivalence

4.0.1. $[\![R]\!]_d^{\leq k} \implies [\![R + k]\!]_d$

Given $(s, \mu, E) \in [\![R]\!]_d^{\leq k}$ we will map using the forward function:

$$(s, \mu, E) \mapsto (\mathsf{forward}(s, E), \mathsf{forward}(\mu, E), \mathsf{forward\_ops}(E))$$

1. By definition, $\mathsf{apply}(d, E) \in \mathcal{L}(R)$
2. Let $d' = \mathsf{apply}(d, E)$
3. Note that $s' = \mathsf{forward}(s, E)$ is span of d'
4. The forward mapping preserves captures moving the indices: $\mu'(x) = \mathsf{forward}(\mu(x), E)$
5. Errors are moving to match the new document, $E' = \mathsf{forward\_ops}(E)$
6. Thus $(s', \mu', E') \in [\![R + k]\!]_d$ because we know that $\mathsf{apply}(d, E) \in \mathcal{L}(R)$, so $(s', \mu') \in [\![R]\!]_d$

4.0.2. $[\![R + k]\!]_d \implies [\![R]\!]_d^{\leq k}$

Given $(s', \mu', E') \in [\![R + k]\!]_d$:

1. we have an $d' = \mathsf{apply}(d, E)$ that meets $(s', \mu') \in [\![R]\!]_{d'}$
2. The inverse mapping recovers original positions:

$$\begin{aligned} s &= \mathsf{backward}(s', E) \\ \mu(x) &= \mathsf{backward}(\mu'(x), E) \\ E' &= \mathsf{backward\_ops}(E) \end{aligned}$$

3. The triple (s, μ, E') satisfies all conditions of $[\![R]\!]_d^{\leq k}$

The equivalence requires:

- Identity preservation: $\mathsf{backward}(\mathsf{forward}(s, E), E) = s$