ICARUS2 Corpus Query Processor Specification

Markus Gärtner

2020

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

The ICARUS2 Corpus Query Processor (ICQP) is a custom evaluation engine for corpus queries that follow the ICARUS2 Query Language (IQL) specification.

Chapter 1

Notations and Definitions

sequence xxx

tree xxx

search node a *search node* n is a tuple (c, δ_p, δ_s) where c is the local constraint predicate taking as argument a target node n_i from a sequence, δ_p is the minimum distance to the node's predecessor and

tree search node a *tree search node* v is a tuple $(c, l,)$

sequence xxx

sequence xxx

sequence xxx

sequence xxx

sequence xxx

1.1 Tree Inclusion

Let T be a *rooted* tree. We say that T is *annotated* if each node v in T is assigned a set of annotations $a_1(v)..a_n(v)$ where each annotation function a_i provides its own alphabet Σ_i of available annotation values. This definition is similar to the basic notion of a *labeled* tree where each node is assigned a single label or value from a common alphabet Σ, but to accommodate the nature of multi-layer annotations in linguistic corpora, we extend this definition. T is further *ordered* if for any node v its children $v_1..v_n$ follow a globally consistent ordering scheme. If not specified otherwise, all trees in this document are *rooted* and *annotated*.

A tree P is said to be *included* in T, denoted $P \sqsubseteq T$, if deleting nodes in T can yield P. Deleting a node v in T means replacing v with the sequence of its children. Solving the tree inclusion problem means determining if P actually is included in T and then also returning all (or up to a specified number of) subtrees of T that include P.

Chapter 2

Plain Matching

Chapter 3

Sequence Matching

Elements participating in sequence matching:

node singular and optionally quantified element

sequence sequence of elements that adhere to given arrangement

grouping sequence of elements that is optionally quantified as a whole

disjunction two or more alternative elements

3.1 Definitions

Let L be a list, then $N_L = |L|$ is its length and $l_i \in L$ denotes the element at position i of the list where $1 \leq i \leq N_L$. Let T be the list of target elements and S the tree of search nodes.

Utility procedures and functions used in the algorithms of this section:

atom(s) Returns the single wrapped child node for s.

child(s, i) Returns the child node s_i on index i for s.

eval(s, t) Evaluates the inner constraints of search node s on the target item t. The result is a cache entry with a boolean value indicating whether the evaluation was successful.

cacheGet(s, t) Retrieve a cached entry for the evaluation of search node s on target element t. If no entry exist, *nil* is returned.

cacheSet(s, t, entry) Stores the given *entry* in the cache for the evaluation of search node s on target element t.

finished() Returns whether or not the state machine has produced a sufficient number of matches for the current target sequence. This function is required for nodes that can produce multiple matches such as *repetirion* and *scan*.

mark(s, t) Stores the fact that s matched t in the preliminary result buffer. Note that each s that models an instance of `iql:IqlNode` is assigned a stack to manage result candidates, so in the case of quantification that allows repetition multiple hits can be stored.

minSize(s) Returns the minimum number of elements needed to satisfy a node.

next(s**)** Returns the next search node in the sequence after s.

size(s**)** Returns the number of child nodes attached to s.

scopeCreate() Registers and returns a scope marker that can later be used to reset all marked matches in the result buffer, back until this marker.

scopeReset(*scope***)** Removes from the result buffer all matches registered via $mark(s, t)$ that have been added since *scope* has been created. This is used by nodes that have to explore many alternative or iterative scenarios, such as *scan*, *repetition* or *branch*.

type(s**)** Returns the type of the given search node, one of the following:[1]

single A single atomic node that is existentially quantified.

repetition A quantifier (c_{min}, c_{max}) and an associated *atom* search node.

negation An embedded node that must **not** match.

universal An embedded node that is expected to match all available target elements. Note that universal quantification is only allowed if the node is the only one in the (resolved) global context.

branch A group of nodes that represent logical alternatives.

scan An embedded node is iteratively checked against all remaining index positions in the target sequence.

region An embedded node and associated index interval $i_{start} = first(s)$ to $i_{end} = last(s)$ of legal positions for matching.

spot An embedded node and a single associated fixed index $i = exact(s)$ for matching.

unmark(s, t**)** Removes the fact that s matched t from the preliminary result buffer.

value(*entry***)** Extracts the actual boolean result value from a cache entry.

3.2 Rules

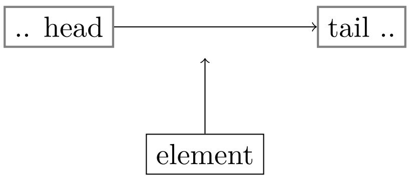
This section describes the recursive rules for constructing the object graph of search nodes from the original elements in the query. All rules assume the existence of already processed nodes that resulted in some *head* part to the left that is connected to a *tail* to the left, as illustrated below:

$$\begin{tikzcd}\boxed{\text{.. head}} \arrow[r] & \boxed{\text{tail ..}}\end{tikzcd}$$

Initially the *head* node is just a generic entry point and the *tail* simply implements the final acceptance node that ensures that all actual search nodes in the query have been evaluated already.

The elements in the query are effectively processed top-down and left-to-right and for every encountered `<IqlElement>` instance a new node is inserted between the current *head* and *tail*:

[1]Nested nodes can be obtained with $atom(s)$ for single embeddings or $child(s, i)$ for indexed elements where $size(s)$ returns the number of embedded nodes and $1 \leq i \leq atom(s)$.



This process is recursive and builds the graph of interconnected search nodes used in the algorithms described in the respective sub sections.

Algorithm 3.1 Match a generic node at a specified position. This procedure exists merely for multiplexing to the specialized counterparts depending on the type of s.

```
 1: procedure MATCH($s, T, j$)
 2:     $type_s \leftarrow$ TYPE($s$)
 3:     if $type_s = single$ then
 4:         return MATCH-NODE($s, T, j$)
 5:     else if $type_s = repetition$ then
 6:         return MATCH-REPETITION($s, T, j$)
 7:     else if $type_s = negation$ then
 8:         return MATCH-NEGATION($s, T, j$)
 9:     else if $type_s = universal$ then
10:         return MATCH-ALL($s, T, j$)
11:     else if $type_s = branch$ then
12:         return MATCH-BRANCH($s, T, j$)
13:     else if $type_s = scan$ then
14:         return MATCH-SCAN($s, T, j$)
15:     else if $type_s = region$ then
16:         return MATCH-REGION($s, T, j$)
17:     else if $type_s = spot$ then
18:         return MATCH-SPOT($s, T, j$)
19:     end if
20: end procedure
```

Actual sequence matching is subsequently performed by calling MATCH($root, T, 1$) with *root* being the first node of the state machine to. This function takes care of multiplexing the call to the specialized procedure depending on a node's type (as provided by *type*(s) in line 2 of Algorithm 3.1).

3.2.1 Single Node

A single instance of `iql:IqlNode` (IQL, Section 1.5.3) is the most basic constraint fragment in an IQL query. Note that in the presence of quantifiers (IQL, Section 1.14) on a node the evaluation is split into two levels: First the `iql:IqlNode` content is processed into a search node s of type *single* and then the quantifiers are handled (which will result in a combination of *repetition*, *branch*, *negation*, *universal* nodes, depending on the complexity of the quantifiers involved), resulting in a potentially very complex node structure with node s as atom.

Algorithm 3.2 Matching of a single node at a specific position. Local constraints and the tail of s are taken into account. Memoization is employed for evaluation of local constraints to prevent repeatedly executing costly constraint expressions.

```
 1: procedure MATCH-NODE($s, T, j$)
 2:     $entry \leftarrow$ CACHEGET($s, t_j$)
 3:     if $entry = nil$ then
 4:         $entry \leftarrow$ EVAL($s, t_j$)
 5:         CACHESET($s, t_j, entry$)
 6:     end if

 7:     $matched \leftarrow$ VALUE($entry$)
 8:     if $matched = true$ then
 9:         $tail \leftarrow next(s)$
10:         $matched \leftarrow$ MATCH($tail, T, j + 1$)
11:     end if
12:     if $matched = true$ then
13:         MARK($s, t_j$)
14:     end if

15:     return $matched$
16: end procedure
```

Matching a single node is pretty straightforward, as shown in Algorithm 3.2. The only local evaluation concerns the execution of the internal constrain expression, for which memoization is employed in order to prevent repeatedly executing it for the same target element. Only if the local constraint evaluates to `true` further evaluation is delegated to the tail of the state machine via matching the $next(s)$ node in line 10. A successful match is automatically stored in the associated result buffer. Note that surrounding *scan* or *repetition* nodes that can produce multiple matches have the ability to reset the state of this result buffer. So if a match fails, the individual nodes won't have to do any cleanup work.

3.2.2 Node Repetition

3.2.3 Existential Negation

3.2.4 Universal Quantification

This special node can only occur as either the root node or as direct element within a disjunction (Section 3.2.5) that in turn is the root. It effectively makes the embedded atom the only node that is allowed to match and it is required to all targets in the current unit-of-interest (UoI). The implementation is very similar to that of scanning (Section 3.2.6) in that it moves the atom through the target sequence and aborting as soon as it fails to match a single element.

3.2.5 Node Disjunction

3.2.6 Iterative Scan

Normally matching a node is done on a single specific position in the target sequence. This covers situations such as node groups with the `ADJACENT` arrangement. But groups that are merely ORDERED (also the default for sequences in IQL when no explicit arrangement is defined) need some sort of scanning mechanism, as individual nodes can be matched at any index position after their predecessor (if one exists). Therefore the *scan* node takes the index argument j of the matcher function as entry point for an iterative search forward that incrementally tries positions until running out of search space or if the result buffer is filled.

Algorithm 3.3 Match an embedded search node at any of the remaining spots in the target sequence, beginning at j.

```
1:  procedure MATCH-SCAN($s, T, j$)
2:      $atom \leftarrow$ ATOM($s$)
3:      $fence \leftarrow N_T -$ MINSIZE($s$) $+1$
4:      $i \leftarrow j$
5:      $result \leftarrow false$
6:      while $i \leq fence \wedge \neg$FINISHED() do          ▷ result limit can end the loop early
7:          $matched \leftarrow$ MATCH($atom, T, i$)
8:          if $matched$ then
9:              $result \leftarrow true$
10:         end if
11:         $i \leftarrow i + 1$
12:     end while
13:     return $result$
14: end procedure
```

Algorithm 3.3 displays the simple algorithm for iterative scanning. Note that this version is not using caching to speed up exploration in the nested *atom* and also only implements the default left-to-right traversal direction. A specialized alternative exists that shifts *atom* through the search space right-to-left and another one that uses caching to skip known dead-ends when matching the nested *atom* (line 7). It is not possible to skip known positive sub-results, as we must still allow nested nodes to register with the result buffer for any such "new" sub-result in the context of a new host node s. When traversing backwards, the algorithm uses the same boundaries, but simply reverses the direction of the loop variable.

3.2.7 Region Interval

All position markers (IQL, Section 3.1) that effectively declare intervals are translated into *region* nodes. Each region node manages one or more index intervals of legal values. Those intervals are refreshed for every UoI at most once. When matching, membership of the current index j to at least one interval is checked first. If this check fails, the entire match call is aborted before the next node is even considered.

3.2.8 Fixed Spot

Node implementation of the markers `IsFirst`, `IsLast` and `IsAt` that all define a fixed index for the target, only influenced by the size of the UoI during matching.

Algorithm 3.4 Match search node only at a fixed location in the target sequence

```
1: procedure MATCH-SPOT($s, T, j$)
2:     $spot \leftarrow$ EXACT($s$)
3:     if $j \neq spot$ then
4:         return $false$
5:     end if
6:     $node \leftarrow$ ATOM($s$)
7:     return MATCH($node, T, j$)
8: end procedure
```

3.3 Algorithm

Algorithm 3.5 Adjacent sequence matching

```
1: procedure MATCH-ADJACENT($C, T, j_{start}, j_{max}$)          ▷ $j_{start} \leq j_{max}, C \subseteq S$
2:     $j \leftarrow j_{start}$
3:     while $j <= j_{max}$ do
4:         if LOCAL_MATCH($s, t_j$) then
5:             return $j$
6:         end if
7:         $j \leftarrow j + 1$
8:     end while
9:     return $-1$
10: end procedure
```

Algorithm 3.6 Quantified matching

```
1: procedure MATCH-QUANTIFIED($s, T, j_{start}, j_{max}$)          ▷ $j_{start} \leq j_{max}$
2:     $j \leftarrow j_{start}$
3:     while $j <= j_{max}$ do
4:         if LOCAL_MATCH($s, t_j$) then
5:             return $j$
6:         end if
7:         $j \leftarrow j + 1$
8:     end while
9:     return $-1$
10: end procedure
```

Chapter 4

Tree Matching

Chapter 5

Complex Markers

IQL defines a wide range of markers for nodes in the query (Chapter 3). The ICQP splits those into three main categories, depending on the way they interact with the target structure and the rest of the state machine:

- **sequence:** The most basic group of markers operates purely on the order of target items, either globally (matching basic tokens) or inside tree structures (matching children of a given parent node). Members of this group are all the positional markers (Section 3.1). Their evaluation can be done very efficiently, but they introduce various helper nodes into the state machine, especially in case of disjunctive marker expressions.

link to node descriptions for gates, borders and clips

- **level:** The argument-less variants (minus the "isAnyGeneration") of tree hierarchy markers ((Section 3.2.1)) form another special group. They operate on the vertical position of a node inside the tree and therefore only look at either the "leaf" property of nodes or the "parent" field of the respective frame in the matcher. Level-based markers are the easiest to evaluate, as they do not rely on any state from utility classes and also do not introduce additional nodes to the state machine.
- **generation:** This group is comprised of all the "xxGeneration" variants of tree hierarchy markers ((Section 3.2.1)) and represents the computationally most expensive markers. To reach the actual nodes to be checked, a subtree has to be traversed in parts or fully. To retain a certain level of efficiency the implementation for these markers uses a simple cache to mark unsuccessfully visited nodes.

5.1 Marker Transformation

Mixed expressions made up of markers of different types can only be handled by the state machine when they are provided in the following way:

1. Disjunctive expressions must be "pure", i.e. they can only contain markers from a single type.
2. Conjunctive expressions must also either be pure themselves or contain only elements that are either direct marker calls or pure expressions.
3. The top-level expression must be in disjunctive normal form (DNF) according to above specifications for conjunctive expressions.

Note that the

Appendices