A Proof of Theorem 1

Theorem 1 *For all rules $r : A \rightarrow C$ and triplestore schemas S,* $\mathbb{I}(\texttt{score}(S,r)) = \mathbb{I}(\texttt{critical}(S,r))$.

Proof: Since the mappings generated by the `critical` and `score` approaches (namely $[\![A]\!]_{\mathbb{C}(S,r)}$ and $[\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$) are post-processed in the same way, we demonstrate the equivalence of the two approaches by demonstrating that ($\Leftarrow$) the set of mappings generated by $\texttt{score}(S,r)$, which are not filtered out by our post-processing of the mappings, is a subset of the mappings generated by $\texttt{critical}(S,r)$ and that ($\Rightarrow$) every mapping $\texttt{critical}(S,r) \setminus \texttt{score}(S,r)$ is redundant with respect to another mapping in $\texttt{critical}(S,r) \cap \texttt{score}(S,r)$. We denote with $\Omega(m,A,S)$ the set of mappings that are not filtered out by our filtering function with respect to A and S. We say that a mapping m is redundant with respect of a set of mappings M (and a schema S and rule r) if the schema consequences computed over M and $' \setminus \{m\}$ are semantically equivalent.

Note that both the critical and the sandbox instance are constructed by taking a triple in schema S^G and substituting its variables. Thus we can associate each triple in the critical or the sandbox instance with the triples in S that can be used to construct them, and we will call the latter the *origin* triples of the former. For each triple in the critical or the sandbox instance, at least one origin triple must exist.

$\Rightarrow$) Let a mapping $m \in [\![A]\!]_{\mathbb{C}(S,r)} \setminus [\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$. We get all triples t_A of A and for each one we will construct a triple t_q of a conjunctive query $q \in \mathbb{Q}(A)$, and mapping m^* of q into $\mathbb{S}(S)$, such that either $m = m^*$ or m^* makes m redundant. For $i \in \tau$, if $t_A[i]$ is a variable $?v$ and $m(?v) = :\lambda$ then set $t_q[i] = ?v$, since there must be a triple in $\mathbb{S}(S)$ with $:\lambda$ in the i^{th} position so if m^* mapped t_q on that triple, $m^*(t_q[i]) = m(t_A[i]) = :\lambda$ (for all triples of the critical instance that have $:\lambda$ in position i, its origin triples have variables in the same position, so the sandbox instance would also have $:\lambda$ in position i). If $t_A[i]$ is a variable $?v$ and $m(?v)$ is a constant c, then we distinguish two cases (a) and (b). Let $t_S \in S$ be an origin triple of $m(t_A)$ in the critical instance, then (a) if $t_S[i] = c$ then $\mathbb{S}(S)$ would have retained c in the corresponding position in the triple of which t_S is the origin, and so we set $t_q[i]$ to $?v$, in order for $(?v \rightarrow c) \in m$ to also belong to mapping m^* of t_q into $\mathbb{S}(S)$, or (b) if $t_S[i]$ is a variable (we know that $:\lambda$ is the element in position i of the triple in the sandbox graph of which t_S is an origin) we consider two sub-cases (b') and ($\neg b'$).

We set $t_q[i]$ to $:\lambda$ in case (b'), namely if there is a position j in a triple $t'_A \in A$ (different position to i, i.e., $j \neq i$ if $t_A = t'_A$), for which $t'_A[j] = t_A[i]$ and for t'_S, an origin triple of t'_A, $t'_S[j] = c$. Condition (b') essentially says that $?v$ will have to be mapped to c in the sandbox graph due to some other position of the rewriting, so $t_q[i]$ can just be set to $:\lambda$ to simply match the corresponding triple in $\mathbb{S}(S)$. In case ($\neg b'$) we set $t_q[i]$ to $?v$; this condition produces a more *general* mapping m^* since $m(?v) = c$ while $m^*(?v)$ will be $:\lambda$. We say that m_2 is more general than m_1, denoted by $m_1 \leq_\lambda m_2$, if $dom(m_1) = dom(m_2)$ and for all $?v \in dom(m_1)$, either $m_2(?v) = m_1(?v)$ or $m_2(?v) = :\lambda$.

Lastly we consider the case of $t_A[i]$ being a constant. If there is an origin triple t_S of $m(t_A)$ in the critical instance such that $t_S[i]$ is the same constant we also set it to $t_q[i]$. Otherwise, if $t_S[i]$ is a variable in any origin triple t_S we set $t_q[i]$ to $:\lambda$. This does not change the mapping under construction m^*. By following the described process for all

triple in A we end up with a rewriting q of A and a mapping m^* from q to $\mathbb{S}(S)$ such that m^* is more general than m.

The existance of the more general mapping m^* makes mapping m redundant. This is true because if m is not filtered out by our post-processing, then m^* would also not be filtered out (since mappings can only be filtered out only when a variable is mapped to a literal, but m^* does not contain any such mapping not already in m). If m is not filtered out, then this would lead to the schema being expanded with triples $m(C)$. The instances of the schema obtained through mapping m are a subset of those obtained through mapping m^*. This can be demonstrated by noticing that for every triple in $m(C)$ there is a corresponding triple in $m^*(C)$ that has at each position, either the same constant c or $:\lambda$, which is then substituted with a variable in the schema consequence. This variable can always be substituted with the same constant c, when instantiating the schema, and therefore there cannot be an instance of the schema consequence generated from mapping m that is not an instance of the schema consequence generated from mapping m^*. In fact, the only case in which a variable $?v$ in a schema cannot be instantiated with a constant c is when c is a literal and $?v$ is in the no-literal set. However, for all mappings $m \in [\![A]\!]_{\mathbb{C}(S,r)} \setminus [\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$, we can notice that whenever our post-processing function adds a variable to the no-literal set, it would have rejected the mapping if that variable was mapped to a literal instead.

$\Leftarrow$) Let a mapping $m^* \in [\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$. This means that there is a $q \in \mathbb{Q}(A)$, for which $m^* \in [\![q]\!]_{\mathbb{S}(S)}$. We get all triples t_q of q and for each one we will show that for the triple t_A in A, that was rewritten into t_q, it holds that either m^* is filtered out by the filtering function Ω, or it is also a mapping from t_A into the critical instance (i.e. $m^*(t_A) \in \mathbb{C}(S,r)$), hence $m^* \in [\![A]\!]_{\mathbb{C}(S,r)}$. This would prove that the mappings generated by $[\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}$ are either subsumed by the mappings in $[\![A]\!]_{\mathbb{C}(S,r)}$, or they would be filtered out anyway, and thus $\Omega([\![\mathbb{Q}(A)]\!]_{\mathbb{S}(S)}, A, S) \subseteq \Omega([\![A]\!]_{\mathbb{C}(S,r)}, A, S)$.

For all $t_q \in q$, for all $i \in \tau$, if $t_q[i]$ is a variable $?v$ then this position has not been changed from rewriting t_A into t_q, thus $?v$ will exist in $t_A[i]$; if $m^*(?v)$ is $:\lambda$, then $m^*(t_q)[i] = :\lambda$, and therefore in any origin triple t_S of $m^*(t_q)$, $t_S[i]$ had to be a variable and so $:\lambda$ will exist in position i in the triples the critical instance will create for t_S, and for these triples m^* will partially map t_A onto them. Similarly, if $t_q[i]$ is a variable $?v$ and $m^*(?v)$ is a constant c, then this constant would be present in $t_S[i]$ and so also in all triples coming from t_S in the critical instance; again m^* will partially map t_A onto the triples in the critical instance generated from t_S.

If $t_q[i]$ is a constant other than $:\lambda$, then the triple t_A that got rewritten to triple t_q would have the same constant in position i. Also, this constant must be present in position i in the triple $m^*(t_q)$ of the sandbox graph, and therefore in position i of any of its origin triples t_S. In fact, by virtue of how the sandbox graph is constructed, any triple of the schema that does not have this constant in position i, cannot be an origin triple of $m^*(t_q)$. Thus again, m^* will partially map t_A onto the triples in the critical instance of which t_S is the origin triple.

Lastly, if $t_q[i]$ is $:\lambda$, then $m^*(t_q)[i] = :\lambda$, and in any origin triple t_S of the sandbox triple $m^*(t_q)$ we have a variable in position i, and in the critical instance we have triples, of which t_S is the origin, with all possible `URI`s or $:\lambda$ in this position; this means that if $m^*(t_A[i])$ is a `URI`, in the triple t_A that got rewritten in t_q, we can "match" it in the

critical instance: if $t_A[i]$ is a variable $?v_1$ then t_A will match the triple in the critical instance (instance of any origin triple t_S of $m^*(t_q)$) that contains $m^*(?v_1)$ in position i (notice that m^* should contain a value for v_1 since all variables are present in all rewritings of A); else, if $t_A[i]$ is a URI there will be a corresponding image triple (again instance of any t_S) in the critical instance with the same constant in this position.

If, however, $m^*(t_A[i])$ is a literal, it is possible that there is no triple in the critical instance that can match the literal $m^*(t_A[i])$ in position i. However we will show that, in this case, the filter function Ω will remove this mapping. If literal $m^*(t_A[i])$ does not occur in position i in any of the triples in the critical instance that have the same origin triple as $m^*(t_q)$ then, by the definition of how the critical instance is constructed, this must be caused by the fact that either (a) $i \in \{1,2\}$ or (b) for all the possible origin triples t_S of $m^*(t_q)$, $t_S[i] \in S^\Delta$ or $t_S[i]$ is a constant other than $m^*(t_A[i])$. In both cases, our filtering function will filter out the mapping. In fact, it filters out mappings where variables are mapped to literals, if they occur in the subject or predicate position of a triple in A. This is true in case (a). Moreover it filters out mappings if a variable in the object position there is a literal l, or a variable mapped to a literal l, such that there is no origin triple t_S of $m^*(t_q)$ such that $t_S[i] = l$ or $t_S[i]$ is a variable not in S^Δ. This is true in case (b). Thus m^* is a mapping from t_A into the critical instance. □

B Proof of Theorem 2

Theorem 2 *For all rules $r : A \rightarrow C$ and triplestore schemas S, $\mathbb{I}(\texttt{score}(S,r)) = \mathbb{I}(r(S))$.*

Theorem 2 states that $\texttt{score}(S,r)$ is semantically equivalent to $r(S)$. We prove this by showing that ($\Rightarrow$) every triple that can be inferred by applying rule r on an instance of S is an instance of the schema $\texttt{score}(S,r) \setminus S$ and that ($\Leftarrow$) every triple in every instance of schema $\texttt{score}(S,r) \setminus S$ can be obtained by applying rule r on an instance of S. To reduce the complexity of the proof, we consider the case without literals (i.e. when no literal occurs in S and r, and all of the variables in the schema S^G are in S^Δ). This proof can be extended to include literals by considering the the post-processing of the mappings, as demonstrated in the proof of Theorem 1. More precisely, we reformulate Theorem 2 as follows:

Lemma 1 *For all rules $r : A \rightarrow C$ and for all triplestore schemas S such that $S^\Delta = vars(S^G)$ and that no literal occurs in r and S^G:*

$\Rightarrow$) for all triple patterns $t^S \in r(S) \setminus S$, for all triples t^I in an instance of t^S, there exists a triple pattern $t_{\texttt{score}} \in \texttt{score}(S,r) \setminus S$ s.t. $\{t^I\} \in \mathbb{I}(\{t_{\texttt{score}}\})$

$\Leftarrow$) for all triple patterns $t_{\texttt{score}} \in \texttt{score}(S,r) \setminus S$, for all triples t^I in an instance of $t_{\texttt{score}}$ there exists a triple pattern $t^S \in r(S) \setminus S$ s.t. $\{t^I\} \in \mathbb{I}(\{t^S\})$.

Proof: $\Rightarrow$) Given the premises of this part of Lemma 1, there must exist a graph $I \in \mathbb{I}(\{S\}$ on which an application of rule r generates t^I. For this to happen, there must exist a mapping $m \in [\![A]\!]_I$, such that $t^I \in m(C)$. Obviously, the set of triples on which A matches into I via m is $m(A) \subseteq I$.

For all triples $t_A^I \in m(A)$ there exists a triple pattern $t_A^S \in S^G$ that models t_A^I (e.g. $\{t_A^I\} \in \mathbb{I}(t_A^S)$). We choose one of such triple pattern t_A^S (in case multiple ones exist) and we call it the *modelling triple* of t_A^I. To prove the first part of Lemma 1 we will make use of the following proposition:

There exists a mapping m^* and a rewriting $q \in \mathbb{Q}(A)$, such that $m^*(q) \subseteq \mathbb{S}(S)$ (thus $m^* \in [\![q]\!]_{\mathbb{S}(S)}$), and that m and m^* agree on all variables except for those that are mapped to $:\lambda$ in m^*. Formally, $dom(m) = dom(m^*)$ and for every variable $?v$ in $dom(m^*)$, either $m^*(?v) = m(?v)$, or $m^*(?v) = :\lambda$.

By proving the proposition above we know that our algorithm would extend the original schema graph S^G with graph pattern $unpack(m^*(C))$, and the set of variables S^Δ with $vars(unpack(m^*(C)))$, where *unpack* is a function that substitutes each $: \lambda$ in a graph with a fresh variable. If we take a triple pattern $c \in C$ such that $t^I = m(\{c\})$, then $unpack(m^*(\{c\}))$ belongs to $\texttt{score}(S,r)^G \setminus S^G$ (because m^* matches a rewriting of the antecedent of the rule r on the sandbox graph). Graph $unpack(m^*(\{c\}))$ also models triple t^I. This is true because for each of the three positions i of the schema triple pattern c, either $c[i]$ is a constant, and therefore this constant will also appear in position i in both $unpack(m^*(\{c\}))[i]$ and $t^I[i]$ or it is a variable, and by our proposition above either (a) $m^*(c)[i] = m(c)[i] = t^I[i]$ and so $unpack(m^*(\{c\}))[i] = t^I[i]$, or (b) $m^*(c)[i] = :\lambda$ and therefore $unpack(m^*(\{c\}))[i]$ is a variable. We can then trivially recreate t^I as an instance of $unpack(m^*(\{c\}))$ by assigning to each variable $?v$ in $unpack(m^*(\{c\}))[i]$ the value $t^I[i]$. Thus our algorithm would generate a triple pattern that models t^I and that would complete the proof of the first part ($\Rightarrow$) of the Theorem.

The proof of our proposition is as follows. Consider every position i in every triple $t_A^I \in m(A)$, and their corresponding modelling triple $t_A^S \in S$. Let t_A be the triple in A such that $m(t_A) = t_A^I$. Let $t_A^{\mathbb{S}}$ be $\mathbb{S}(t_A^S)$, and thus $t_A^{\mathbb{S}} \in \mathbb{S}(S)$. We are now going to construct a query q, rewriting of $\mathbb{Q}(A)$, such that $[\![q]\!]_{\mathbb{S}(S)}$ gives us the mapping m^* in our proposition. And to do this, we have to construct a triple pattern t_q which will be the rewriting of t_A in q. By definition of how our rewritings are constructed, every element $t_q[i]$ is either $t_A[i]$ or $:\lambda$.

To set the value of each element in the triple pattern t_q we consider the four possible cases that arise from $t_A[i]$ and $t_A^S[i]$ being either a constant or a variable. In parallel, we consider the element-by-element evaluation of q on $\mathbb{S}(S)$ which generates m^*.

1. If $t_A[i]$ and $t_A^S[i]$ are both constants, then since $t_A^I \in m(A)$, it must be true that $t_A[i] = t_A^I[i]$. Moreover, since t_A^S must be a model of t_A^I, it follows that $t_A^S[i] = t_A^I[i]$ and therefore $t_A^{\mathbb{S}}[i] = t_A^I[i]$. We set element $t_q[i]$ to be the constant $t_A[i]$, which matches the constant $t_A^{\mathbb{S}}[i]$.
2. If $t_A[i]$ is a constant but $t_A^S[i]$ is a variable, then we know that $t_A^{\mathbb{S}}[i] = :\lambda$. We set element $t_q[i]$ to be the constant $:\lambda$, which matches the constant $t_A^{\mathbb{S}}[i]$.
3. If $t_A[i]$ is a variable $?v$ and $t_A^S[i]$ is a constant x, then we know that $t_A^I[i] = t_A^{\mathbb{S}}[i] = x$. Therefore mapping m must contain binding $?v \rightarrow x$. We set element $t_q[i]$ to be the variable $?v$, so that when we complete the construction of q, m^* will contain mapping $?v \rightarrow x$.
4. If $t_A[i]$ and $t_A^S[i]$ are both variables, it must be that $t_A^{\mathbb{S}}[i] = :\lambda$. If it exists a triple $t'_A \in A$ and a modelling triple $t_A^{S\prime}$ of $m(t'_A)$, and position j such that $t'_A[i]$ is variable $?v$ and $t_A^{S\prime}[i]$ is a constant, then we set element $t_q[i]$ to be the constant $:\lambda$ ($t_A^{\mathbb{S}}[i]$ will

be $:\lambda$ in our sandobox graph). Note that even though we don't use variable $t'_A[i]$ in this part of our rewriting, the aforementioned existence of triple t'_A will force m^* to contain a binding for this variable, and this binding will be in m because this pair $t'_A[i]$ and $t^{S\prime}_A[i]$ will be treated by case 3. Otherwise we set element $t_q[i]$ to be the variable $t_A[i]$, which will bind with the value $:\lambda$ of $t^{\mathbb{S}}_A[i]$ generating binding $?v \rightarrow :\lambda$.

This proves our proposition, as we have constructed a query q, which belongs to $\mathbb{Q}(A)$, which matches $\mathbb{S}(S)$ and generates a mapping $m*$ which agrees on m on all variables, except for those which m^* maps to $:\lambda$.

$\Leftarrow$) Given the premises of the second direction of Lemma 1, there must exist a mapping m^* and a query $q \in \mathbb{Q}(A)$ such that: $t_{\texttt{score}} \in \mathit{unpack}(m^*(C))$, $m^*(q) \subseteq \mathbb{S}(S)$ and $\mathit{dom}(m^*) = \mathit{dom}(A)$. Note that $\mathbb{S}(S)$ is an instance of S. There must exist a triple pattern t_C in C such that $\mathit{unpack}(m^*(\{t_C\})) = t_{\texttt{score}}$.

Note that *unpack* transforms a triple in a triple pattern by changing occurrences of λ into fresh variables, if no λ occurs in the triple, the triple remains unchanged.

For all positions i of t_C, if $t_C[i]$ is a variable $?v$, given that $\mathit{unpack}(m^*(\{t_C\}))$ models t^I, then mapping m^* either contains the binding $?v \rightarrow t^I[i]$, or it contains $?v \rightarrow :\lambda$. Now consider the mapping m generated by modifying mapping m^* in the following way. For each position i of t_C, if $t_C[i]$ is a variable $?v$ and $?v \rightarrow :\lambda \in m^*$, substitute this binding with $?v \rightarrow t^I[i]$. It is trivial to see that $t^I \in m(C)$. We will now show that this mapping m can be computed by evaluating the original query A on a specific instance of S by showing that $m(A)$ is an instance of S. By virtue of how query q is constructed, we know that all of its elements in all of its triples must be the same as A, or the constant $:\lambda$ instead.

As an intermediate step, we show that $m(q)$ is an instance of S. Since we know that $m^*(q)$ is an instance of S, and that $:\lambda$ cannot occur in the triples in S, every occurrence of $:\lambda$ in $m^*(q)$ must be generated from a variable $?v$ in a triple in S. Since each variable in S^G occurs only once, we can change any occurrence of $:\lambda$ in $m^*(q)$ into any other constant and still have an instance of S, since, intuitively, in the corresponding positions S contains different variables. Given that the difference between m and m^* is that for some variables that m^* binds to $:\lambda$, m binds them to a different constant, it is easy to see that the $m(q)$ can be generated from $m^*(q)$ only by changing some occurrences of λ in its triples into a different constant. Thus $m(q)$ is an instance of S.

Now we are going to show that $m(A)$ is an instance of S. For all triples $t_A \in A$, our query rewriting algorithm has produced a $t^I_A \in a$ for which, for each position i, we know that $t^I_A[i] = t_A[i]$ or $t^I_A[i] = :\lambda$. Obviously for $m(t^I_A) \in m(q)$, the same holds, i.e., for every position i, $m(t^I_A)[i] = m(t_A)[i]$ or $m(t^I_A)[i] = :\lambda$. Since $m(t^I_A)$ is an instance of S, there must exist a mapping γ and a triple pattern $t^{\mathbb{S}}_A \in S^G$ such that $\gamma(\{t^{\mathbb{S}}_A\}) = m(t^I_A)$. Since $t^{\mathbb{S}}_A$ cannot contain $:\lambda$, for every position i where $m(t^I_A)[i] = :\lambda$, $t^{\mathbb{S}}_A[i]$ contains a variable $?v_i$ which does not occur anywhere else in S^G (by virtue of variables occurring only once in the schema); we can obtain the mapping from $t^{\mathbb{S}}_A$ to $m(t_A)$ by substituting each such binding $?v_i \rightarrow :\lambda$ with $?v_i \rightarrow m(t_A)[i]$ in γ. For all triples $t_A \in A$, $m(t_A)$ is an instance of S, and therefore $\bigcup_{t_A \in A}\{m(t_A)\}$, which equals $m(A)$, is an instance of S. $\square$

C Translation of SHACL constraints into Triplestore Schemas

In this section of the appendix we present an approach to convert a subset of SHACL constraints into triplestore schemas and vice versa. An implementation of our approaches is available in our code repository.[1]

C.1 Translation from SHACL constraints to Triplestore Schemas

We present a simple approach to translate SHACL constraints, captured by a set of shapes, into triplestore schemas. For purpose of readability, we present it as a set of templates of SHACL shapes and their corresponding translations in Table 2. We use constants `:a`, `:b`, `:c`, `:p`, `:q`, `:s`, `:k` and `"m"` as placeholders constants and "..." to represent a repetition of elements. The set of SHACL terms that we consider is listed in Figure 1. This approach does not deal with all possible shapes constructed with with these terms, and extensions of this approach are possible. For example, it does not deal with cardinality constraints with cardinality different from 1, or recursive shapes.

`sh:NodeShape`	`sh:IRIOrLiteral`	`sh:class`
`sh:targetObjectsOf`	`sh:in`	`sh:hasValue`
`sh:targetSubjectsOf`	`sh:property`	`sh:node`
`sh:targetClass`	`sh:path`	`sh:or`
`sh:nodeKind`	`sh:inversePath`	`sh:not`
`sh:IRI`	`sh:minCount`	

Fig. 1: SHACL terms considered in this approach.

The proposed approach to translate a set of SHACL constraints into a triplestore schema consists of three phases. First, we initialise an empty schema $S = \langle \varnothing, \varnothing, \varnothing \rangle$, and an empty set of uninstantiable predicates U. Second, then modify S on a shape-by-shape basis by applying the required transformations as specified in Table 2. For each SHACL shape s, this involves finding a matching template in the table, namely a template that can be transformed into s by a substitution of the placeholder constants, and then applying the transformations corresponding to that template under the same substitution. Lastly, we remove all triples from S^G whose predicate is in U, which we would have populated as we apply the transformations.

We consider three transformations. Given a set of existential constraints E, the transformation *include*(E) simply adds constraints E to $S^{\exists}$. Transformations *allow*(T^p, T^{Δ}) and *restrict*(T^p, T^{Δ}) are defined for a set of triple patterns T^p, where all the triples in T^p have the same predicate p, and a no literal set T^{Δ}. Transformations *allow*(T^p, T^{Δ}) and *restrict*(T^p, T^{Δ}) add T^p and T^{Δ} to S^G and S^{Δ}, respectively, if S^G does not contain a triple pattern with p. If S^G contains a set of triple patterns P with p as predicate, transformation *restrict*(T^p, T^{Δ}) removes from S patterns P and their corresponding variables P^{Δ} from the no-literal set; then adds to S the "schema intersection" graph G'^p and its no literal set G'^{Δ} where $\mathbb{I}(\langle G'^p, G'^{\Delta}, \varnothing \rangle) = \mathbb{I}(P, P^{\Delta}, \varnothing) \cap \mathbb{I}(T^p, T^{\Delta}, \varnothing)$. If the schema intersection computed by the *restrict* transformation is empty, then this predicate cannot occur in the instances of S and we add it to a set of uninstantiable predicates U.

[1] https://github.com/paolo7/ISWC2019-code

For example, let us imagine a schema $S = \langle\{[\texttt{?v1}, \texttt{:a}, \texttt{?v2}]\}\}, \{\texttt{?v1}, \texttt{?v2}\}, S^{\exists}\rangle$, that is, triples with predicate `:a` are allowed in the instances of S if they have an `IRI` as object. Then consider the application of this transformation, *restrict*($\{[\texttt{:b}, \texttt{:a}, \texttt{?v3}]\}, \{\}$) which restricts triples with predicate `:a` to have `:b` as subject. The result of this transformation is $\langle\{[\texttt{:b}, \texttt{:a}, \texttt{?v4}]\}\}, \{?v4\}, S^{\exists}\rangle$, namely the schema of triples with predicates `:a` which have `:b` as subject and a `IRI` as an object.

SHACL	Triplestore Schema

`:s rdf:type sh:NodeShape;`
`  sh:targetObjectsOf :p;`
`  sh:nodeKind sh:IRIOrLiteral .`
	allow({ `?v1 :p ?v2` }, {`?v1`})

`:s rdf:type sh:NodeShape;`
`  sh:targetObjectsOf :p;`
`  sh:nodeKind sh:IRI .`
	restrict({ `?v1 :p ?v2` }, {`?v1, ?v2`})

`:s rdf:type sh:NodeShape;`
`  sh:targetObjectsOf :p;`
`  sh:in (:a "m" ... ) .`
	restrict({`?v1 :p :a` ,
	`?v2 :p "m"` ,
	`?v3 :p ...` }, {`?v1, ?v2, ?v3`})

`:s rdf:type sh:NodeShape;`
`  sh:targetSubjectsOf :p;`
`  sh:in (:a :b ... ) .`
	restrict({`:a :p ?v1` ,
	`:b :p ?v2` ,
	`... :p ?v3` }, {})

`:s rdf:type sh:NodeShape;`
`  sh:targetObjectsOf :p;`
`  sh:class :c .`
	restrict({ `?v1 :p ?v2` }, {`?v1, ?v2`})
	allow({ `?v1 rdf:type ?2` }, {`?v1`})
	include(`?v1 :p ?v2` $\rightarrow^{\exists}$ `?v2 rdf:type :c`)

`:s rdf:type sh:NodeShape;`
`  sh:targetSubjectsOf :p;`
`  sh:class :c .`
	allow({ `?v1 :p ?v2` }, {`?v1`})
	allow({ `?v1 rdf:type ?v2` }, {`?v1`})
	include(`?v1 :p ?v2` $\rightarrow^{\exists}$ `?v1 rdf:type :c`)

`:s rdf:type sh:NodeShape;`
`  sh:targetClass :c;`
`  sh:property [`
`  sh:path :p;`
`  sh:minCount 1; ] .`
	allow({ `?v1 rdf:type ?v2` }, {`?v1`})
	allow({ `?v1 :p ?v2` }, {`?v1`})
	include(`?v1 rdf:type :c` $\rightarrow^{\exists}$ `?v1 :p ?v2`)

`:s rdf:type sh:NodeShape;`
`  sh:targetObjectsOf :p;`
`  sh:property [`
`  sh:path :q;`
`  sh:minCount 1; ] .`
	restrict({ `?v1 :p ?v2` }, {`?v1, ?v2`})
	allow({ `?v1 :q ?v2` }, {`?v1`})
	include(`?v1 :p ?v2` $\rightarrow^{\exists}$ `?v2 :q ?v3`)

`:s rdf:type sh:NodeShape;`
`  sh:targetSubjectsOf :p;`
`  sh:property [`
`  sh:path :q;`
`  sh:minCount 1; ] .`
	allow({ `?v1 :p ?v2` }, {$v1$})
	allow({ `?v1 :q ?v2` }, {$v1$})
	include(`?v1 :p ?v2` $\rightarrow^{\exists}$ `?v1 :q ?v3`)

:s rdf:type sh:NodeShape; sh:targetClass :c; sh:property [sh:path [sh:inversePath :p]; sh:minCount 1;] .	*allow*({?v1 rdf:type ?v2}, {?v1}) *allow*({?v1 :p ?v2}, {?v1}) *include*(?v1 rdf:type :c $\rightarrow^{\exists}$?v2 :p ?v1)
:s rdf:type sh:NodeShape; sh:targetObjectsOf :p; sh:property [sh:path [sh:inversePath :q]; sh:minCount 1;] .	*restrict*({?v1 :p ?v2}, {?v1, ?v2}) *allow*({?v1 :q ?v2}, {?v1}) *include*(?v1 :p ?v2 $\rightarrow^{\exists}$?v3 :q ?v2)
:s rdf:type sh:NodeShape; sh:targetSubjectsOf :p; sh:property [sh:path [sh:inversePath :q]; sh:minCount 1;] .	*allow*({?v1 :p ?v2}, {*v*1}) *allow*({?v1 :q ?v2}, {*v*1}) *include*(?v1 :p ?v2 $\rightarrow^{\exists}$?v3 :q ?v1)
:s rdf:type sh:NodeShape; sh:targetClass :c; sh:property [sh:path :p; sh:hasValue :a; sh:minCount 1;] .	*allow*({?v1 rdf:type ?v2}, {?v1}) *allow*({?v1 :p ?v2}, {?v1}) *include*(?v1 rdf:type :c $\rightarrow^{\exists}$?v1 :p :a)
:s rdf:type sh:NodeShape; sh:targetObjectsOf :p; sh:property [sh:path :q; sh:hasValue :a; sh:minCount 1;] .	*restrict*({?v1 :p ?v2}, {?v1, ?v2}) *allow*({?v1 :q ?v2}, {?v1}) *include*(?v1 :p ?v2 $\rightarrow^{\exists}$?v2 :q :a)
:s rdf:type sh:NodeShape; sh:targetSubjectsOf :p; sh:property [sh:path :q; sh:hasValue :a; sh:minCount 1;] .	*allow*({?v1 :p ?v2}, {*v*1}) *allow*({?v1 :q ?v2}, {*v*1}) *include*(?v1 :p ?v2 $\rightarrow^{\exists}$?v1 :q :a)
:s rdf:type sh:NodeShape; sh:targetClass :c; sh:property [sh:path [sh:inversePath :p]; sh:hasValue :a; sh:minCount 1;] .	*allow*({?v1 rdf:type ?v2}, {?v1}) *allow*({?v1 :p ?v2}, {?v1}) *include*(?v1 rdf:type :c $\rightarrow^{\exists}$:a :p ?v1)
:s rdf:type sh:NodeShape; sh:targetObjectsOf :p; sh:property [sh:path [sh:inversePath :q]; sh:hasValue :a; sh:minCount 1;] .	*restrict*({?v1 :p ?v2}, {?v1, ?v2}) *allow*({?v1 :q ?v2}, {?v1}) *include*(?v1 :p ?v2 $\rightarrow^{\exists}$:a :q ?v2)

:s rdf:type sh:NodeShape;
  sh:targetSubjectsOf :p;
  sh:property [
  sh:path [sh:inversePath :q];
  sh:hasValue :a;
  sh:minCount 1;] .
	allow({?v1 :p ?v2}, {*v*1})
	allow({?v1 :q ?v2}, {*v*1})
	include(?v1 :p ?v2 $\rightarrow^{\exists}$:a :q ?v1)

:s rdf:type sh:NodeShape;
  sh:targetNode :a;
  sh:property [
  sh:path :p;
  sh:node :k;] .
:k rdf:type sh:NodeShape;
  sh:property [
  sh:path :q;
  sh:hasValue :b;] .
	allow({?v1 :p ?v2}, {*v*1})
	allow({?v1 :q ?v2}, {*v*1})
	include(:a :p ?v1 $\rightarrow^{\exists}$?v1 :q :b)

:s rdf:type sh:NodeShape;
  sh:targetNode :a;
  sh:property [
  sh:path
  [sh:inversePath :p];
  sh:node :k;] .
:k rdf:type sh:NodeShape;
  sh:property [
  sh:path :q;
  sh:hasValue :b;] .
	allow({?v1 :p ?v2}, {*v*1})
	allow({?v1 :q ?v2}, {*v*1})
	include(?v1 :p :a $\rightarrow^{\exists}$?v1 :q :b)

:s rdf:type sh:NodeShape;
  sh:targetNode :a;
  sh:property [
  sh:path :p;
  sh:node :k;] .
:k rdf:type sh:NodeShape;
  sh:property [
  sh:path
  [sh:inversePath :q];
  sh:hasValue :b;] .
	allow({?v1 :p ?v2}, {*v*1})
	allow({?v1 :q ?v2}, {*v*1})
	include(:a :p ?v1 $\rightarrow^{\exists}$:b :q ?v1)

:s rdf:type sh:NodeShape;
  sh:targetNode :a;
  sh:property [
  sh:path
  [sh:inversePath :p];
  sh:node :k;] .
:k rdf:type sh:NodeShape;
  sh:property [
  sh:path
  [sh:inversePath :q];
  sh:hasValue :b;] .
	allow({?v1 :p ?v2}, {*v*1})
	allow({?v1 :q ?v2}, {*v*1})
	include(?v1 :p :a $\rightarrow^{\exists}$:b :q ?v1)

:s rdf:type sh:NodeShape;
  sh:targetNode :a;
  sh:or (
  [sh:property [
  sh:path :p;
  sh:node :k;]
  ]
  [sh:property [
  sh:path :p;
  sh:node [
  sh:not [
  sh:in (:b)]] ;]
  ]) .
:k rdf:type sh:NodeShape;
  sh:property [
  sh:path
  [sh:inversePath :q] ;
  sh:minCount 1 ;] .
	allow({?v1 :p ?v2}, {$v1$})
	allow({?v1 :q ?v2}, {$v1$})
	include(:a :p :b $\rightarrow^{\exists}$?v1 :q :b)

:s rdf:type sh:NodeShape;
  sh:targetNode :a;
  sh:or (
  [sh:property [
  sh:path :p;
  sh:node :k;]
  ]
  [sh:property [
  sh:path :p;
  sh:node [
  sh:not [
  sh:in (:b)]] ;]
  ]) .
:k rdf:type sh:NodeShape;
  sh:property [
  sh:path
  [sh:inversePath :q] ;
  sh:hasValue :c;] .
	allow({?v1 :p ?v2}, {$v1$})
	allow({?v1 :q ?v2}, {$v1$})
	include(:a :p :b $\rightarrow^{\exists}$:c :q :b)

Table 2: Templates of SHACL shapes and their translation transformations. To reduce the size of this table we have omitted some templates that can be obtained from the ones above by simply switching the subject and object of triple patterns.

C.2 Translation form Triplestore Schemas to SHACL constraints

Given a schema $S : \langle S^G, S^\Delta, S^\exists \rangle$ we reconstruct a corresponding set of SHACL shapes in two phases. In the first phase, we use Table 2 in the opposite direction than in the previous section, to deal with existential rules. More specifically, we match each existential constraint in $S^\exists$ with the existential constraints of Table 2, and re-create the corresponding SHACL shape.

Unfortunately we cannot use the table in a similar fashion to translate the graph pattern and no literal sets in S, as S^G and S^Δ might not correspond to a single template. Instead, they might be a result of the interactions of multiple such templates. Therefore to compute the corresponding SHACL shapes of S^G and S^Δ we present Algorithm 3 and its helper algorithms 1 and 2. Algorithm 3 computes the `SHACL` graph for a schema S on a predicate by predicate basis. The SHACL translation of graph pattern S^G and no literal set S^Δ can then be computed by executing Algorithm 3 for each predicate p in S^G. In the algorithms we use notation $e \ll e'$ to indicate that an element e is subsumed by another element e'. This condition is true if and only if one of these conditions apply: (1) $e = e'$ (2) e' is a variable that allows for literals or (3) e is a `URI` and e' is a variable.

Algorithm 1 Helper algorithm to convert a set of elements e into `SHACL` constraints.

procedure COMPUTE_CONSTRAINTS(e, S^Δ)
  $r \leftarrow$ empty set of triples
  $b, b' \leftarrow$ a new fresh `URI` each
  if e contains a variable not in the no-literal set **then**
    `add` $\langle b$, `sh:nodeKind`, `sh:IRIOrLiteral`$\rangle$ to r
  else
    if e contains a variable in the no-literal set **then**
      `add` $\langle b$, `sh:nodeKind`, `sh:IRI`$\rangle$ to r
    if e contains constants **then**
      $s \leftarrow$ `SHACL` list of constants in e
      `add` $\langle b'$, `sh:in`, $s\rangle$ to r
  return r

Algorithm 2 Helper algorithms to add constraints C to a shape i

procedure ADD_CONSTRAINTS(i, C)
  $r \leftarrow$ empty set of triples
  $B \leftarrow$ `SHACL` list of $\{t[1] \mid t \in C\}$
  `add` $\langle i$, `sh:or`, $B\rangle$ to r
  return r

Algorithm 3 Compute the SHACL triples for predicate p and schema $\langle S^G, S^\Delta \rangle$

procedure COMPUTESHAPE(p,S^G,S^Δ)
  $r \leftarrow$ empty set of triples
  for each $t \in S^G$ s.t. $t[1]$ or $t[3]$ is a constant **do**
    $i \leftarrow$ a new fresh `URI`
    add $\langle i$, `rdf:type`, `sh:NodeShape`$\rangle$ to r
    if $t[1]$ is a `URI` **then**
      add $\langle i$, `sh:targetNode`, $t[1]\rangle$ and $\langle i$, `sh:path`, $p\rangle$ to r
      $e \leftarrow \{t'[3] \mid \forall\, t' \in S^G$s.t. $t[1] \ll t'[1]\}$
      $E \leftarrow$COMPUTE_CONSTRAINTS(e, S^Δ)
      add ADD_CONSTRAINTS(i, E) to r
  if $\nexists\, t \in S^G$s.t. $t[1]$ is a variable **then**
    $i \leftarrow$ a new fresh `URI`
    add $\langle i$, `rdf:type`, `sh:NodeShape`$\rangle$ and $\langle i$, `sh:targetSubjectsOf`, $p\rangle$ to r
    $e \leftarrow \{t'[1] \mid \forall\, t' \in S^G$s.t. $t'[1]$is constant$\}$
    $E \leftarrow$COMPUTE_CONSTRAINTS(e, S^Δ)
    add ADD_CONSTRAINTS(i, E) to r
  else
    $i, i' \leftarrow$ a new fresh `URI` each
    add $\langle i$, `rdf:type`, `sh:NodeShape`$\rangle$ and $\langle i$, `sh:targetObjectsOf`, $p\rangle$ to r
    $C \leftarrow \{t[3] \mid t \in S^G$s.t. $t[1]$is a variable$\}$
    $E \leftarrow$COMPUTE_CONSTRAINTS(C, S^Δ)
    for each $t \in S^G$ s.t. $t[1]$ is a constant **do**
      for each $t' \in S^G$ s.t. $t'[3] \ll t[3]$ **do**
        $D \leftarrow \{t''[1] \mid t'' \in S^G$s.t. $t'[3] \ll t''[3]\}$
        $b, b', b'' \leftarrow$ a new fresh `URI` each
        add $\langle b$, `rdf:type`, `sh:NodeShape`$\rangle$, $\langle b$, `sh:property`, $b'\rangle$, $\langle b'$, `sh:path`, $b''\rangle$ and $\langle b''$, `sh:inversePath`, $p\rangle$ to r
        $H \leftarrow$ COMPUTE_CONSTRAINTS(D, S^Δ)
        add ADD_CONSTRAINTS(b', H) to r
        $n \leftarrow$ the only triple in COMPUTE_CONSTRAINTS($\{t'[3]\}, S^\Delta$)
        add n and $\langle n[1]$, `sh:node`, $b\rangle$ to E
    add ADD_CONSTRAINTS(i, E) to r
    $E' \leftarrow$ empty set of triples
    add $\langle i'$, `rdf:type`, `sh:NodeShape`$\rangle$ and $\langle i'$, `sh:targetSubjectsOf`, $p\rangle$ to r
    for each $t \in S^G$ s.t. $t[1]$ or $t[3]$ is a constant **do**
      $n \leftarrow$ the only triple in COMPUTE_CONSTRAINTS($\{t'[1]\}, S^\Delta$)
      add n to E'
      if $t[3]$ is not a variable v s.t. $v \notin s^\Delta$ **then**
        $b, b' \leftarrow$ a new fresh `URI` each
        add $\langle b$, `rdf:type`, `sh:NodeShape`$\rangle$, $\langle b$, `sh:property`, $b'\rangle$ and $\langle b'$, `sh:path`, $p\rangle$ to r
        $H \leftarrow$ empty set of triples
        if $t[3]$ is a variable **then**
          $H \leftarrow$ COMPUTE_CONSTRAINTS($\{t[3]\}, S^\Delta$)
        else
          $b'' \leftarrow$ a new fresh `URI`
          $H \leftarrow \{\langle b''$, `sh:hasValue`, $t[3]\rangle\}$
        add ADD_CONSTRAINTS(b', H) to r
        add $\langle n[1]$, `sh:node`, $b\rangle$ to E'
    add ADD_CONSTRAINTS(i', E') to r
  return r