

ADAPTIVE NON-LINEAR PATTERN MATCHING AUTOMATA

RICK ERKENS AND MAURICE LAVEAUX

Eindhoven University of Technology, De Groene Loper 5, 5612 AE, Eindhoven, The Netherlands
e-mail address: r.j.a.erkens@tue.nl
e-mail address: m.laveaux@tue.nl

ABSTRACT. Efficient pattern matching is fundamental for practical term rewrite engines. By preprocessing the given patterns into a finite deterministic automaton the matching patterns can be decided in a single traversal of the relevant parts of the input term. Most automaton-based techniques are restricted to linear patterns, where each variable occurs at most once, and require an additional post-processing step to check so-called variable consistency. However, we can show that interleaving the variable consistency and pattern matching phases can reduce the number of required steps to find all matches. Therefore, we take the existing adaptive pattern matching automata as introduced by Sekar et al and extend these with consistency checks. We prove that the resulting deterministic pattern matching automaton is correct, and show several examples where some reduction can be achieved.

1. INTRODUCTION

Term rewriting is a universal model of computation that is used in various applications, for example to evaluate equalities or simplify expressions in model checking and theorem proving. In its simplest form, a binary relation on *terms*, which is described by the *term rewrite system*, defines the available reduction steps. Term rewriting is then the process of repeatedly applying these reduction steps when applicable. The fundamental step in finding which reduction steps are applicable is *pattern matching*.

There are two variants for the pattern matching problem. *Root pattern matching* can be described as follows: given a term t and a set of patterns, determine the subset of patterns such that these are (syntactically) equal to t under a suitable substitution for their variables. The other variant, called *complete pattern matching*, determines the matching patterns for all subterms of t . Root pattern matching is often sufficient for term rewriting, because applying reduction steps can make matches found for subterms obsolete. A root pattern matching algorithm can also be used to naively solve the complete pattern matching problem by applying it to every subterm.

As the matching patterns need to be decided at each reduction step, various *term indexing* techniques [SRV01] have been proposed to determine matching patterns efficiently. An adaptive pattern matching automaton [SRR95], abbreviated as APMA (plural: APMAs), is a tree-like data structure that is constructed from a set of patterns. By using such

Key words and phrases: Pattern matching, Term indexing, Tree automata.

an automaton one can decide the matching patterns by only examining each function symbol of the input term at most once. Moreover, it allows for *adaptive* strategies, *i.e.*, matching strategies that are fixed before construction such as the left-to-right traversal of the automata in [Grä91]. The size of an APMA is worst-case exponential in the number of patterns. However, in practice its size is typically smaller and this construction step is beneficial when many terms have to be matched against a fixed pattern set.

The APMA approach works for sets of linear patterns, that is, in every pattern every variable occurs at most once. As mentioned in other literature [Grä91, SRR95] the non-linear matching problem can be solved by first preprocessing the patterns, then solving the linear matching problem and lastly checking so-called *variable consistency*. We can show that performing matching and consistency checking separately does not minimise the amount of steps required to find all matches. Therefore, we extend the existing APMA to perform consistency checking as part of the matching process. Our extension preserves the adaptive traversal of [SRR95] and allows information about the matching step to influence the consistency checking, and the other way around. The influence of consistency checking on the matching step is only beneficial in a setting where checking (syntactic) term equality, which is necessary for consistency checking, can be performed in constant time. This is typically the case in systems where (sub)terms are maximally shared, which besides constant time equality checks also has the advantage of a compact representation of terms.

We introduce the notion of *consistency automata*, abbreviated as CA (plural: CAs), to perform the variable consistency check efficiently for a set of patterns. The practical use of these automata is based on similar observations as the pattern matching automata. Namely, there may be overlapping consistency constraints for multiple patterns in a set. We prove the correctness for these consistency automata and provide an analysis of its time and space complexity. We prove that the consistency automaton approach yields a correct consistency checking algorithm for non-linear patterns. Finally, we introduce *adaptive non-linear pattern matching automata* (ANPMAs), a combination of adaptive pattern matching automata and consistency automata. ANPMAs use information from both match and consistency checks to allow the removal of redundant steps. We show that ANPMAs yield a correct matching algorithm for non-linear patterns. To this end we also give a correctness proof for the APMA approach from [SRR95], which was not given in the original work.

1.1. Structure of the paper. In Figure 1 there is a simple example ANPMA for the pattern set $\{\ell_1 : f(x, x), \ell_2 : f(a, b), \ell_3 : f(a, a)\}$. It has edges labelled with function symbols

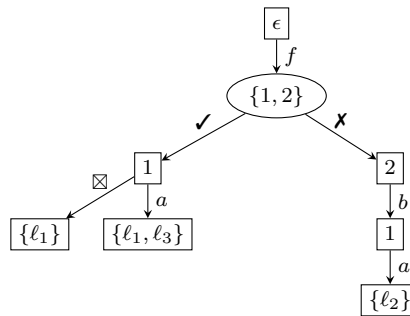


FIGURE 1. An example ANPMA.

(coming from states labelled with positions) and edges labelled with \checkmark or \times . In Section 5 we define ANPMAs formally. Since we will treat a correctness proof, we first discuss both elements of ANPMAs.

- In Section 3 we recall APMAs from [SRR95]. We focus on formalities and give a correctness proof of this method which was not in the original work. Correctness intuitively means that the automaton that is constructed for a set of linear patterns, is suitable to efficiently decide the matching problem. We also show that there are no redundancies in APMA evaluation. That is, there is no state such that the same outgoing transition is taken, no matter what the input term is.
- In Section 4 we define CAs, again with a focus on formalities and correctness proofs. Given a set of patterns, a set of consistency partitions can be computed. The automaton that is constructed for this set of partitions, is suitable to efficiently decide the variable consistency problem. Although some redundancies can be removed, it is still difficult to remove all redundancies from CAs.

These two sections provide a proper foundation for the formal details of ANPMAs in Section 5, since ANPMAs are automata that can have an interleaving between APMA states (with outgoing function symbol transitions) and CA states (with outgoing \checkmark - and \times -transitions). We give the construction algorithm, a correctness proof based on the proofs of the previous section, and show that ANPMAs are at least as efficient as first applying APMAs and then consistency checking, or the other way around.

It is difficult to obtain an ANPMA without any redundancies. We give some examples of pattern sets that are benefitted by consistency checks, but where it is difficult to define a general construction procedure that can make use of these benefits. Lastly we state the optimisation problem for ANPMAs such that it can be picked up for future work.

1.2. Related Work. We compare this work with other term indexing techniques. Most techniques use tree-like data structures with deterministic [Car84, Grä91, SRR95, vW07] or non-deterministic [FM01, Chr93, McC92, Vor95, Gra95] evaluation. In this setting a deterministic evaluation guarantees that all positions in the input term are inspected at most once. Non-deterministic approaches typically have smaller automata, but the same position might be inspected multiple times for input terms as a result of backtracking.

Most of the mentioned techniques do not support matching non-linear patterns directly. *Discrimination trees* [McC92], *substitution trees* [Gra95] can be extended with on-the-fly consistency checks for matching non-linear patterns. However, their evaluation strategy is restricted to left-to-right evaluation and variable consistency must be checked whenever a variable which has already been bound occurs again at the position that is currently inspected in the term. Our approach of introducing a state to check term equality is also present in *match trees* [vW07], *code trees* [Vor95] and the decision trees used in Dedukti [HB20]. The main advantage of ANPMAs is that consistency checks are allowed to occur at any point in the automaton, the evaluation strategy is not limited to a fixed strategy and there are fewer redundant checks, which makes the matching time shorter.

2. PRELIMINARIES

In this section the preliminaries of first-order terms and the root pattern matching problem are defined. We denote the *disjoint union* of two sets A and B by $A \uplus B$. Given two sets

A and B we use $A \rightarrow B$, $A \dashrightarrow B$ and $A \leftrightarrow B$ to denote the sets of total, partial and total injective functions from A to B respectively. We assume that a partial function yields a special symbol \perp for elements in its domain for which it is undefined. Given a function $f : A \rightarrow B$ we use $f[a \mapsto b]$ to denote the mapping that satisfies $f(x) = f[a \mapsto b](x)$ if $x \neq a$ and $f(x) = b$ if $x = a$.

Let $\mathbb{F} = \biguplus_{i \in \mathbb{N}} \mathbb{F}_i$ be a *ranked* alphabet. We say that $f \in \mathbb{F}_i$ is a *function symbol* with arity, written $\text{ar}(f)$, equal to i . Let $\Sigma = \mathbb{V} \uplus \mathbb{F}$ be a *signature* where \mathbb{V} is a set of variables. The set of terms over Σ , denoted by \mathbb{T}_Σ , is defined as the smallest set such that $\mathbb{V} \subseteq \mathbb{T}_\Sigma$ and whenever $t_1, \dots, t_n \in \mathbb{T}_\Sigma$ and $f \in \mathbb{F}_n$, then also $f(t_1, \dots, t_n) \in \mathbb{T}_\Sigma$. We typically use the symbols x, y for variables, symbols a, b for function symbols of arity zero (constants), f, g, h for function symbols of other arities and t, u for terms. The *head* of a term, written as head , is defined as $\text{head}(x) = x$ for a variable x and $\text{head}(f(t_1, \dots, t_n)) = f$ for a term $f(t_1, \dots, t_n)$. We use $\text{vars}(t)$ to denote the set of variables that occur in term t . A term for which $\text{vars}(t) = \emptyset$ is called a *ground term*. A *pattern* is a term of the form $f(t_1, \dots, t_n)$. A pattern is *linear* iff every variable occurs at most once in it.

We define the (syntactical) equality relation $= \subseteq \mathbb{T}^2$ as the smallest relation such that $x = x$ for all $x \in \mathbb{V}$, and $f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)$ if and only if $t_i = t'_i$ for all $1 \leq i \leq n$. Furthermore, the equality relation modulo variables $=_\omega \subseteq \mathbb{T}^2$ is the smallest relation such that $x =_\omega y$ for all $x, y \in \mathbb{V}$, and $f(t_1, \dots, t_n) =_\omega f(t'_1, \dots, t'_n)$ if and only if $t_i =_\omega t'_i$ for all $1 \leq i \leq n$. Both $=$ and $=_\omega$ satisfy reflexivity, symmetry and transitivity and thus are equivalence relations, and we can observe that $= \subseteq =_\omega$.

A *substitution* σ is a total function from variables to terms. The application of a substitution σ to a term t , denoted by t^σ , is the term where variables of t have been replaced by the term assigned by the substitution. This can be inductively defined as $x^\sigma = \sigma(x)$ and $f(t_1, \dots, t_n)^\sigma = f(t_1^\sigma, \dots, t_n^\sigma)$. We say that term u *matches* t , denoted by $t \leq u$, iff there is a substitution σ such that $t^\sigma = u$. Terms t and u *unify* iff there is a substitution σ such that $t^\sigma = u^\sigma$.

We define the set of *positions* \mathbb{P} as the set of finite sequences over natural numbers where the *root* position, denoted by ϵ , is the identity element and concatenation, denoted by dot, is an associative operator. Given a term t we define $t[\epsilon] = t$ and if $t[p] = f(t_1, \dots, t_n)$ then $t[p.i]$ for $1 \leq i \leq n$ is equal to t_i . Note that $t[p]$ may not be defined, e.g., $f(x, y)[3]$ and $f(x, y)[1.1]$ are not defined. A position p is *higher* than q , denoted by $p \sqsubseteq q$, iff there is position $r \in \mathbb{N}^*$ such that $p.r = q$. Position p is *strictly higher* than q , denoted by $p \sqsubset q$, whenever $p \sqsubseteq q$ and $p \neq q$. We say that a term $t[q]$ is a *subterm* of $t[p]$ if $p \sqsubset q$ and $t[q]$ is defined. The replacement of the subterm at position p by term u in term t is denoted by $t[p/u]$, which is defined as $t[\epsilon/u] = u$ and $f(t_1, \dots, t_n)[(i.p)/u] = f(t_1, \dots, t_i[p/u], \dots, t_n)$. The *fringe* of a term t , denoted by $\mathcal{F}(t)$, is the set of all positions at which a variable occurs, given by $\mathcal{F}(t) = \{p \in \mathbb{P} \mid t[p] \in \mathbb{V}\}$.

We also define a restricted signature for terms with a one-to-one correspondence between variables and positions. First, we define $\mathbb{V}_\mathbb{P}$ as the set of *position variables* $\{\omega_p \mid p \in \mathbb{P}\}$. Consider the signature $\Sigma_\mathbb{P} = \mathbb{F} \uplus \mathbb{V}_\mathbb{P}$. We say that a term $t \in \mathbb{T}_{\Sigma_\mathbb{P}}$ is *position annotated* iff for all $p \in \mathcal{F}(t)$ we have that $t[p] = \omega_p$. For example, the terms ω_ϵ and $f(\omega_1, g(\omega_{2.1}))$ are position annotated, whereas the terms $f(x)$ and $f(\omega_{1.1})$ are not. Position annotated patterns are linear as each variable can occur at most once.

A *matching function* decides for a given term and a set of patterns the exact subset of these patterns that match the given term.

Definition 2.1. Let $\mathcal{L} \subseteq \mathbb{T}_\Sigma$ be a set of patterns. A function $match_{\mathcal{L}} : \mathbb{T}_\Sigma \rightarrow 2^{\mathbb{T}_\Sigma}$ is a *matching function for \mathcal{L}* iff for all terms t we have $match_{\mathcal{L}}(t) = \{\ell \in \mathcal{L} \mid \exists \sigma : \ell^\sigma = t\}$. If \mathcal{L} is a set of linear patterns then $match_{\mathcal{L}}$ is a *linear matching function*.

3. ADAPTIVE PATTERN MATCHING AUTOMATA

For a single linear pattern to match a given term it is necessary that every function symbol of the pattern occurs at the same position in the given term. This can be decided by a single traversal of the input pattern.

Proposition 3.1. *Let ℓ be linear pattern and t any term. We have that $\ell \leq t$ if and only if for all positions p : if $head(\ell[p]) \in \mathbb{F}$ then $head(\ell[p]) = head(t[p])$.*

For linear patterns a naive matching algorithm follows directly from this proposition: to find all matches for term t one can check the requirement stated on positions in the proposition for every pattern separately. However, for a *set* of linear patterns we can observe that whenever a specific position of the given term is inspected, a decision can be made for all patterns at the same time. Exploiting these kind of observations to yield an efficient decision procedure is the purpose of so-called *term indexing techniques* [SRV01]. Sekar et al. [SRR95] describe the construction of a so-called *adaptive pattern matching automaton*, abbreviated as APMA. Given a set of *linear* patterns \mathcal{L} an APMA can be constructed that can be used to decide for every term $t \in \mathbb{T}_\Sigma$ which patterns of \mathcal{L} are matches for t . The main advantage of an APMA over other indexing techniques and the naive approach is that every position of any input term is inspected at most once.

As an introduction to the techniques that are developed in later sections, we recall the evaluation and construction procedures of APMAs. The presentation that we use is slightly more formal compared to the presentation by Sekar et al. The extra formalities provide a more convenient foundation for our extensions. Moreover we present a correctness proof that did not appear in the original work, which also mainly serves as a stepping stone to the correctness proofs for our extensions.

APMAs are state machines in which every state is a *match* state, which is labelled with a position, or *final* state, which is labelled with a set of patterns. Match states indicate that the term under evaluation is being inspected at the labelled position. Final states indicate that a set of matching patterns is found. The transitions are labelled by function symbols or an additional *fresh* symbol $\boxtimes \notin \mathbb{F}$. Let $\mathbb{F}_{\boxtimes} = \mathbb{F} \uplus \{\boxtimes\}$.

Definition 3.2. An APMA is a tuple (S, δ, L, s_0) where:

- $S = S_M \uplus S_F$ is a finite set of states consisting of a set of *match states* S_M and a set of *final states* S_F ;
- $\delta : S_M \times \mathbb{F}_{\boxtimes} \rightarrow S$ is a partial transition function;
- $L = L_M \uplus L_F$ is a state labelling function with $L_M : S_M \rightarrow \mathbb{P}$ and $L_F : S_F \rightarrow 2^{\mathbb{T}_\Sigma}$; and
- $s_0 \in S_M$ is the initial state.

We only consider APMAs that have a tree structure that is rooted in s_0 . That is, δ is an injective partial mapping and there is no pair (s, f) with $\delta(s, f) = s_0$.

We illustrate the evaluation of an APMA by means of an example. Consider the patterns $f(a, b, x)$, $f(c, b, x)$ and $f(c, b, c)$ with $a, b, c \in \mathbb{F}_0$, $f \in \mathbb{F}_3$ and $x \in \mathbb{V}$. Figure 2 shows an APMA that can be used to decide which of these patterns match for any given term. Every

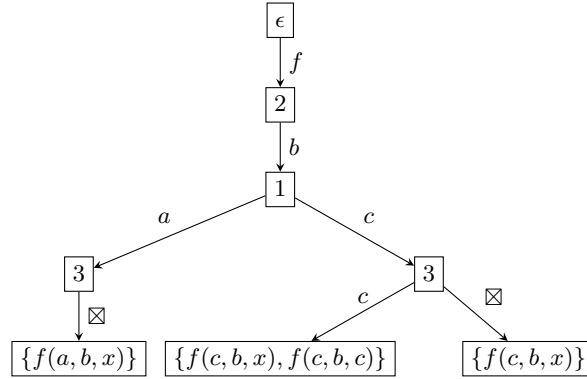


FIGURE 2. An APMA constructed from the patterns $f(a, b, x)$, $f(c, b, x)$ and $f(c, b, c)$ with $a, b, c \in \mathbb{F}_0$, $f \in \mathbb{F}_3$ and $x \in \mathbb{V}$.

state with outgoing transitions is a match state, and the other states at the bottom are the final states. The match states are labelled with the position that is inspected during the evaluation of that state.

The evaluation of an APMA for a given term is defined in the function `MATCH` defined in Algorithm 1. Upon reaching a final state $s \in S_F$ the evaluation yields the set of terms $L_F(s)$, because these patterns match by construction. In a match state $s \in S_M$ the head symbol $\text{head}(t[L_M(s)])$ is examined. If there is an outgoing transition labelled with the examined head symbol then evaluation continues with the reached state; otherwise the \boxtimes -transition is taken. Whenever there is also no outgoing \boxtimes -transition then there is no match by construction and the evaluation returns the empty set as a result.

Algorithm 1 Given a state s of the APMA $M = (S, \delta, L, s_0)$ and a term t , this algorithm computes the pattern matches for t by evaluating M on t .

$$\text{MATCH}(M, t, s) = \begin{cases} L_F(s) & \text{if } s \in S_F \\ \text{MATCH}(M, t, \delta(s, f)) & \text{if } s \in S_M \wedge \delta(s, f) \neq \perp \\ \text{MATCH}(M, t, \delta(s, \boxtimes)) & \text{if } s \in S_M \wedge \delta(s, \boxtimes) \neq \perp \wedge \delta(s, f) = \perp \\ \emptyset & \text{if } s \in S_M \wedge \delta(s, \boxtimes) = \delta(s, f) = \perp \end{cases}$$

where $f = \text{head}(t[L_M(s)])$

Consider the APMA M of Figure 2 and let the initial state s_0 be the topmost state in the figure. We have $\text{MATCH}(M, f(a, b, a), s_0) = \text{MATCH}(M, f(a, b, a), \delta(s_0, f)) = \dots = \{f(a, b, x)\}$. Similarly, we derive that $\text{MATCH}(M, f(b, b, b), s_0) = \emptyset$ and that evaluating term $f(c, b, b)$ yields the pattern set $\{f(c, b, x)\}$.

The construction procedure for APMAs is defined in Algorithm 2. We use ‘:=’ to denote assignments to variables, and we use $M[S := S']$ to denote that the element S of the tuple M gets updated to the value S' . For example, $M := M[S_F := S_F \cup \{s\}]$ means that s is added to the set of final states in the APMA M .

Intuitively, the function `CONSTRUCT` constructs the APMA from the root state to final states based on the set of patterns \mathcal{L} that could still result in a match and a selection function $\text{SELECT} : 2^{\mathbb{P}} \rightarrow \mathbb{P}$. For convenience we assume that the patterns in \mathcal{L} are position-annotated

and as such these patterns are also linear. In later sections we drop this assumption and treat arbitrary (non-linear) patterns. The algorithm is initially called with the initial state s_0 , after which every recursive call corresponds to a state deeper in the tree. The parameter **SELECT** is a function that determines in each recursive call which position from **work** becomes the label for the current state. Based on the selected position, the current state and the pattern set, outgoing transitions are created to fresh states where the construction continues recursively.

The parameter **pref** denotes the *prefix* for a state s . The function symbols in **pref** represent which function symbols have been matched so far and the variables in **pref** represent which positions have not been inspected yet. The special symbol \boxtimes is used to indicate that none of the patterns have the function symbol of the given term at that position. For example, the prefix $f(\omega_1, \omega_2, \omega_3)$ represents that f occurs at position ϵ of the input term and the variables at positions 1, 2 and 3 encode that these positions have not been inspected yet, or equivalently that these subterms are unknown. The prefix can be reconstructed by following the transitions from s_0 to s .

Each recursive call starts by removing all the patterns from \mathcal{L} that do not unify with **pref**. Any match for the removed patterns cannot reach this state of the subautomaton that is currently being constructed. Therefore, the removed patterns do not have to be considered for the remainder of the construction. If **pref** has the symbol \boxtimes at position p then none of the patterns in \mathcal{L} that have a non-variable subterm at position p can unify with the prefix any more, because \boxtimes does not occur in the patterns. If there are no variables in **pref** then there is nothing to be inspected anymore. This is the termination condition for the construction; the current state s will be labelled with the patterns that unify with **pref**. Otherwise, the work that still has to be done, *i.e.*, the set of positions that still have to be inspected, is the fringe of **pref**, denoted by $\mathcal{F}(\text{pref})$.

Algorithm 2 Given a finite set of patterns \mathcal{L} , this algorithm constructs an APMA for \mathcal{L} . Initially, it is called with $M = (\emptyset, \emptyset, \emptyset, s_0)$, the initial state $s = s_0$ and the prefix **pref** = ω_ϵ .

```

1: procedure CONSTRUCT( $\mathcal{L}$ , SELECT,  $M$ ,  $s$ , pref)
2:    $\mathcal{L}' := \{\ell \in \mathcal{L} \mid \ell \text{ unifies with } \text{pref}\}$ 
3:   work :=  $\mathcal{F}(\text{pref})$ 
4:   if work =  $\emptyset$  then
5:      $M := M[S_F := (S_F \cup \{s\}), L_F := L_F[s \mapsto \mathcal{L}']]$ 
6:   else
7:     pos := SELECT(work)
8:      $M := M[S_M := (S_M \cup \{s\}), L_M := L_M[s \mapsto \text{pos}]]$ 
9:      $F := \{f \in \mathbb{F} \mid \exists \ell \in \mathcal{L}' : \text{head}(\ell[\text{pos}]) = f\}$ 
10:    for  $f \in F$  do
11:       $M := M[\delta := \delta[(s, f) \mapsto s']]$  where  $s'$  is a fresh unbranded state w.r.t.  $M$ 
12:       $M := \text{CONSTRUCT}(\mathcal{L}, \text{SELECT}, M, s', \text{pref}[\text{pos}/f(\omega_{\text{pos}.1}, \dots, \omega_{\text{pos.ar}(f)})])$ 
13:    if  $\exists \ell \in \mathcal{L}' : \exists \text{pos}' \sqsubseteq \text{pos} : \text{head}(\ell[\text{pos}']) \in \mathbb{V}$  then
14:       $M := M[\delta := \delta[(s, \boxtimes) \mapsto s']]$  where  $s'$  is a fresh unbranded state w.r.t.  $M$ 
15:       $M := \text{CONSTRUCT}(\mathcal{L}, \text{SELECT}, M, s', \text{pref}[\text{pos}/\boxtimes])$ 
16:  return  $M$ 

```

3.1. Proof of Correctness. We prove that this construction yields an APMA that is suitable to solve the matching problem for non-empty finite sets of linear patterns. Meaning that the evaluation of the constructed APMA for any term t is a linear matching function as defined in Definition 2.1.

We make use of the following auxilliary definitions. A *path* to s_n is a sequence of state and function symbol pairs $(s_0, f_0), \dots, (s_{n-1}, f_{n-1}) \in S_M \times \mathbb{F}_{\boxtimes}$ such that $\delta(s_i, f_i) = s_{i+1}$ for all $i < n$. Because δ is required to be an injective partial mapping there is a unique path to s for every state s , which we denote by $\text{path}(s)$. A match state s is *top-down* iff $L(s) = \epsilon$ or there is a pair (s_i, f_i) in $\text{path}(s)$ with $L(s_i).j = L(s)$ for some $1 \leq j \leq \text{ar}(f_i)$. State s is *canonical* iff there are no two states in $\text{path}(s)$ that are labelled with the same position. Finally we say that an APMA is *well-formed* iff all match states are top-down and canonical.

Well-formed APMAs allow us to inductively reconstruct the prefix of a state s as it was created in the construction algorithm. We allow slight overloading of the notation and denote the prefix of state s by $\text{pref}(s)$. It is constructed inductively for well-formed APMAs by $\text{pref}(s_0) = \omega_\epsilon$ and if $\delta(s_i, f) = s_{i+1}$ then $\text{pref}(s_{i+1}) = \text{pref}(s_i)[L(s_i)/f(\omega_{L(s_i).1}, \dots, \omega_{L(s_i).\text{ar}(f)})]$. Similarly, we denote the patterns of state s by $\mathcal{L}(s) = \{\ell \in \mathcal{L} \mid \ell \text{ unifies with } \text{pref}(s)\}$ for all states. Lastly we use an arbitrary function $\text{SELECT} : 2^{\mathbb{P}} \rightarrow \mathbb{P}$ such that for all sets of positions work we have $\text{SELECT}(\text{work}) \in \text{work}$.

Lemma 3.3. *For all finite, non-empty sets of patterns \mathcal{L} we have that the procedure $\text{CONSTRUCT}(\mathcal{L}, \text{SELECT}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon)$ terminates and yields a well-formed APMA $M = (S, \delta, L, s_0)$.*

Proof. Since \mathcal{L} is finite, the set F that is computed on line 9 is also finite. Therefore the for loop only treats finitely many function symbols. It remains to show that the recursion terminates. The prefixes of the recursive calls are ordered by the strict matching ordering $<$. Observe that whenever $\text{pref}[p]$ is defined, there must be a pattern $\ell \in \mathcal{L}$ such that $\ell[p]$ is defined as well. There are only finitely many positions defined by the patterns of \mathcal{L} . Therefore $<$ is a well-founded ordering on the recursive calls, which guarantees termination.

Upon termination the result M is indeed an APMA. For every function symbol in F exactly one transition is created and at most one \boxtimes -transition is created, so δ is a partial mapping. Since the target states of these transitions are fresh we have that δ is injective. Moreover there is no transition to s_0 since the algorithm is initially called with s_0 . Hence M is an APMA.

We check that M is well-formed. By construction we have $L(s_0) = \epsilon$ since the construction procedure is called with the prefix ω_ϵ . Let s be an arbitrary non-final state and consider the stage of the construction algorithm $\text{CONSTRUCT}(\mathcal{L}, \text{SELECT}, M, s, \text{pref})$. A position label $p.i$ is only chosen if it occurs in the fringe of pref . Therefore there must have been a state labelled with p where the variable $\omega_{p.i}$ was put in the prefix, so s must be top-down. Lastly s is canonical because once a position p is chosen, it cannot be chosen again since the variable ω_p is replaced by an element of \mathbb{F}_{\boxtimes} in the prefix. Hence M meets all requirements for well-formedness. \square

For the remainder of the correctness proof assume an arbitrary finite, non-empty set of position annotated patterns \mathcal{L} and let $M = (S, \delta, L, s_0)$ be the APMA for \mathcal{L} that results from $\text{CONSTRUCT}(\mathcal{L}, \text{SELECT}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon)$.

The following lemma states some claims about final states. They are mostly necessities for the other lemmas. A final state is characterised by a ground prefix and a non-empty set of patterns.

Lemma 3.4. *For every every final state s : (a) the set $L(s)$ is non-empty, (b) $\text{pref}(s)$ is a ground term, and (c) for all $\ell \in L(s)$ we have $\ell \leq \text{pref}(s)$. Moreover (d) for every pattern $\ell \in \mathcal{L}$ there is at least one final state s with $\ell \in L(s)$.*

Proof. First observe that $\mathcal{L}(s)$ is non-empty for all states s . Let s be a final state.

- a) Since $L(s) = \mathcal{L}(s)$ and $\mathcal{L}(s)$ is non-empty the claim holds.
- b) The prefix $\text{pref}(s)$ is ground for final states s because the construction only creates final states if $\text{pref}(s)$ has no variables.
- c) By construction we have $L(s) = \{\ell \in \mathcal{L} \mid \ell \text{ unifies with } \text{pref}(s)\}$. Since $\text{pref}(s)$ is ground we have that for all $\ell \in L(s)$ that $\ell \leq \text{pref}(s)$.
- d) Let $\ell \in \mathcal{L}$. The following invariant holds for constructed APMAs. For all match states s' , if $\ell \in \mathcal{L}(s')$ then there exist an f and an s'' such that $\delta(s', f) = s''$ and $\ell \in \mathcal{L}(s'')$. From the fact that $\mathcal{L}(s_0) = \mathcal{L}$ it then follows that every pattern will end up in some final state. \square

Lemma 3.5 is an invariant that relates the construction algorithm to the evaluation function MATCH. It means that whenever t matches ℓ , then ℓ will invariantly be in the set of patterns that is associated with the states that are visited by MATCH. The proof is an induction on the length of the path to the state under consideration. The details of this proof can be found in the appendix.

Lemma 3.5. *Let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$. For all states s such that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$ it holds that $\mathcal{L}_t \subseteq \mathcal{L}(s)$.*

Lemma 3.6 claims two straightforward correctness properties. Firstly, if no pattern matches t , then the evaluation function MATCH will yield the empty set of patterns. Secondly, if at least one pattern matches t , then the evaluation function will reach a final state. The proof details are in the appendix.

Lemma 3.6. *Let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$.*

- a) *If $\mathcal{L}_t = \emptyset$ then $\text{MATCH}(M, t, s_0) = \emptyset$;*
- b) *If $\mathcal{L}_t \neq \emptyset$ then $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_f)$ for some final state s_f .*

From the invariant claimed in Lemma 3.5, it follows that all pattern matches of t are returned by the evaluation. The following lemma additionally claims the converse: all patterns returned by the evaluation are indeed pattern matches for t . A detailed proof can be found in the appendix.

Lemma 3.7. *Let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$. If $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_f)$ for some final state s_f then $L(s_f) = \mathcal{L}_t$.*

From these lemmas the following correctness theorem follows.

Theorem 3.8. *The function $\lambda t. \text{MATCH}(M, t, s_0)$ is a linear matching function for pattern set \mathcal{L} .*

Proof. Let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$. If $\mathcal{L}_t = \emptyset$ then by Lemma 3.6 we get that $\text{MATCH}(M, t, s_0) = \emptyset = \mathcal{L}_t$ as required. If \mathcal{L}_t is non-empty then by Lemma 3.6 we have that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_f)$ for some final state s_f . Then by definition of MATCH we get $\text{MATCH}(M, t, s_f) = L(s_f)$. From Lemma 3.7 it follows that $L(s_f) = \mathcal{L}_t$, by which we can conclude $\text{MATCH}(M, t, s_0) = \mathcal{L}_t$. Hence $\lambda t. \text{MATCH}(M, t, s_0)$ is a linear matching function for \mathcal{L} . \square

3.2. Redundancy. Algorithm 2 follows a very simple kind of construction. At every state, figure out what still needs to be observed and then choose one of the positions from `work`. Sekar et al. already observed that one kind of redundancy can be completely removed from the computed set `work`. If there is a position in `work` where no pattern in \mathcal{L} has a function symbol at that position, then there is nothing worthwhile to observe. In such cases Algorithm 2 creates no outgoing function symbol transitions, but it does create a \boxtimes -transition. However, by definition of `MATCH` this transition is then taken regardless of the input term. The evaluation of this state always takes an unnecessary step. In this case we call the state \boxtimes -redundant. Formally, we identify two types of redundancies.

Definition 3.9 (Redundancy for match states). Let s be a match state and let $M = (S, \delta, L, s_0)$ be an APMA.

- Given $f \in \mathbb{F}_{\boxtimes}$, we say that s is *f-redundant* iff for all terms t , whenever $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$, then $\text{MATCH}(M, t, s) = \text{MATCH}(M, t, \delta(s, f))$.
- We say that s is *dead* iff for all terms t , whenever $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$, then $\text{MATCH}(M, t, s) = \emptyset$.

In Figure 2 the leftmost state labelled by position 3 is \boxtimes -redundant. The procedure `MATCH` will always take the \boxtimes -transition upon reaching that state. Avoiding this kind of redundancies reduces the number of steps needed to declare a match, which yields a more efficient matching algorithm. Algorithm 2 only allows \boxtimes -redundancies, as described above, so this notion of *f-redundancy* is more general than needed for this section. In Section 5 we will see that *f-redundant* states, with $f \in \mathbb{F}$, can occur due to interleaving with consistency checks. To avoid redundancies, Sekar et al. included the following in their construction procedure.

Lemma 3.10. *Consider Algorithm 2 where Line 3 is replaced by*

$$\text{work} := \mathcal{F}(\text{pref}) \setminus \bigcap_{\ell \in \mathcal{L}'} \{p \in \mathbb{P} \mid \text{head}(\ell[p]) \notin \mathbb{F}\}.$$

Then the correctness argument of Theorem 3.8 still applies, and every state of every APMA resulting from the construction is not dead and not f-redundant, for every f.

Proof. For correctness we only point out the one reparation that needs to be done to the proof of Lemma 3.4. It no longer holds that $\text{pref}(s)$ is ground for every final state s . So it only remains to show that $\ell \leq \text{pref}(s)$ for all final states. The prefix of a final state can still have variables, but then for every position $p \in \mathcal{F}(\text{pref}(s))$ the pattern $\ell \in \mathcal{L}'$ has a variable at that position or a higher position. This means that we still have $\ell \leq \text{pref}(s)$ for every $\ell \in L(s)$.

To show that there are no redundancies, let s be a match state of M and let t be a term such that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$. Since M is canonical, we know that $L(s) \neq L(s')$ for all states s' in $\text{path}(s)$. No position is observed twice, thus we only need to observe non-redundancy locally per state.

- If s has at least one outgoing *f*-transition with $f \neq \boxtimes$ then by definition $\text{MATCH}(M, t, s) = \text{MATCH}(M, t, \delta(s, f))$. In this case we can easily construct a term t' with $t \neq t'$ and $\text{MATCH}(M, t', s_0) = \text{MATCH}(M, t', s)$ such that one of the following holds:
 - $\text{MATCH}(M, t', s) = \text{MATCH}(M, t', \delta(s, g))$, for some $g \in \mathbb{F}$ with $f \neq g$;
 - $\text{MATCH}(M, t', s) = \text{MATCH}(M, t', \delta(s, \boxtimes))$, if there is a \boxtimes -transition from s
 - $\text{MATCH}(M, t', s) = \emptyset$, if s has no outgoing transition save for the *f*-transition.

In all three cases the evaluation of t' continues differently, so s is not f -redundant.

- Otherwise, if s has no outgoing f -transition, then it could have an outgoing \boxtimes -transition and no other transitions. By the definition of CONSTRUCT this can only occur if no pattern in $\mathcal{L}(s)$ has a function symbol on position $L(s)$. However, the alternative CONSTRUCT defined in this lemma ensures that the chosen position for $L(s)$ is not a part of $\text{work}(s)$ and therefore this case cannot occur.
- Otherwise, if s has no outgoing transitions at all, then it must be that s is dead. This case can only occur if $\mathcal{L}(s)$ is empty, which is impossible. \square

So in the simple case of linear terms, an APMA can be constructed that has no redundancies. In later sections we see that avoiding redundancies for the non-linear matching algorithm is difficult.

3.3. Strategies. In this section we recall some of the known results on the time and space complexity of APMAs [SRR95]. These results also carry over to the extensions that we propose later on. The reason for providing a selection function in the construction procedure is to define a strategy for the order in which the positions of the input term are inspected. The chosen strategy influences both the size and the matching time of the resulting automaton. For the size of an automaton it is quite natural to count the number of states. For example, consider the APMA for the patterns in Figure 2 where position one was chosen before position two. The resulting APMA has nine states instead of eight.

There are heuristics described for the selection function based on local strategies, which only look at the prefix and the pattern set during the construction presented in [SRR95]. For example taking the position with the highest number of alternative function symbols in the patterns to maximise the breadth of the automaton. However, these heuristics can all be shown to be inefficient in certain cases [SRR95]. Due to the following result it is unlikely that optimal choices can be made locally.

It was already known that minimisation of the number of states is NP-complete [SRR95]. This result is based on the NP-completeness of minimising the number of states for Trie indexing structures [CS76]. Alternatively, one can consider the *breadth* of the resulting automaton, which is given by the number of final states, as the measurement of size. The reason for this is that the total number of states is at most bounded by the height of the automaton times the number of final states. The upper bound on breadth for the optimal selection function for a set of patterns $\{\ell_0, \dots, \ell_n\}$ is $\mathcal{O}(\prod_{i=0}^n |\ell_i|)$ [SRR95]. The lower bound on breadth for any selection function is $\mathcal{O}(\alpha^{n-1})$, where α is the average number of function symbols and n the number of patterns.

The matching time of an APMA can be defined in two different ways. We can consider a notion of *average matching time* based on a distribution of input terms. However, this information is not typically available and assuming a uniform distribution seems unrealistic in practice. Therefore, we consider the upper bound on the matching time to be determined by the final state with the highest depth for the optimal selection function, and the lower bound is the lowest depth for any selection function. Here, the depth of a state is defined by the length of the path to the root of the automaton. For any pattern sets the upper bound on the depth is S , where S is the total number of non-variable positions, and the lower bound on the depth is $\Omega(S)$ [SRR95].

An alternative approach is to define a notion of relative efficiency that compares two APMA's. For a given APMA M and a given term t we define the matching time as the

evaluation depth, denoted by $\text{ED}(M, t)$, based on the number of recursive MATCH calls required to reach a final state. Then one can consider a notion of relative efficiency that compares two APMAs.

Definition 3.11. Given two APMAs $M = (S, \delta, L, s_0)$ and $M' = (S', \delta', L', s'_0)$ for a set of patterns P . We say that $M \preceq M'$ iff for all terms $t \in \mathbb{T}_\Sigma$ it holds that $\text{ED}(M, t) \leq \text{ED}(M', t)$.

An interesting observation for the selection function is that choosing so-called *index* positions always yields a more efficient APMA as in the definition above and in the number of states. These index positions are positions where all the patterns that can still match (the set \mathcal{L} in the construction algorithm) have a function symbol. As such, taking these index positions (if they are available) always yields the optimal choice. Furthermore, avoiding redundant states can also be shown to always yield a more efficient APMA. Aside from these observations, two selection functions can easily yield APMAs that are incomparable with respect to \preceq . A number of heuristics for selection functions can be found in [Mar08].

4. CONSISTENCY AUTOMATA

A linear matching algorithm can be used to solve the non-linear matching problem by renaming the patterns and checking so-called *variable consistency* after the matching phase [Grä91, SRV01, SRR95]. As a preprocessing step a renaming procedure is applied. It renames each pattern to a linear pattern by introducing new variables. The variable consistency check ensures that the newly introduced variables which correspond to variables in the non-linear pattern can be assigned a single value in the matching substitution. This can be seen as a many-to-one context where multiple introduced variables correspond to a single variable in the original pattern. For the non-linear matching algorithm we can first use a linear matching algorithm to determine matches for the renamed patterns. Followed by a consistency check to remove the linear patterns for which the matching substitution is not valid for the original patterns.

4.1. Pattern Renaming. A straightforward way to achieve the renaming would be to introduce new variables for each position in the fringe of each pattern. However, for patterns $f(x, a)$ and $f(x', y')$ the variables x and x' could be identical such that the assignment for x (or equally x') yields a substitution for both patterns. We can use position annotated variables for this purpose, which are identical for the same position in different patterns, to obtain these overlapping assignments.

For the consistency check it is necessary to keep track of equality constraints between variables that correspond to a single variable before a non-linear pattern is renamed. For this purpose we use the notion of *consistency classes* [SRV01]. A consistency class is a set of positions that are expected to be equal in order to yield a match.

Definition 4.1. Given a term t and a consistency class $C \subseteq \mathbb{P}$ we say that t is *consistent* with respect to C if and only if $t[p] = t[q]$ for all $p, q \in C$.

A pattern can give rise to multiple consistency classes. For instance, consider the pattern $f(x, x, y, y, y, z)$. Based on the occurrences of variables x, y and z we derive the three consistency classes $\{1, 2\}$, $\{3, 4, 5\}$ and $\{6\}$. This means that for the input term $t = f(t_1, \dots, t_6)$ that $t[1] = t[2]$ and $t[3] = t[4] = t[5]$ must hold, and finally $t[i] = t[i]$ holds trivially for all $1 \leq i \leq 6$, for this term to be consistent w.r.t. these classes. A set of disjoint

consistency classes is referred to as a *consistency partition*. The notion of term consistency w.r.t. a consistency class is extended as follows. A term t is consistent with respect to a consistency partition P iff t is consistent with respect to C for every $C \in P$.

First, we illustrate the renaming procedure by means of an example. Consider three patterns $f(x, x, z)$, $f(x, y, x)$ and $f(x, x, x)$. After renaming we obtain the following pairs of a linear pattern and the corresponding consistency partition: $(f(\omega_1, \omega_2, \omega_3), P_1)$, $(f(\omega_1, \omega_2, \omega_3), P_2)$ and $(f(\omega_1, \omega_2, \omega_3), P_3)$; with the consistency partitions $P_1 = \{\{1, 2\}, \{3\}\}$, $P_2 = \{\{1, 3\}, \{2\}\}$ and $P_3 = \{\{1, 2, 3\}\}$. The term $f(a, a, b)$ matches $f(\omega_1, \omega_2, \omega_3)$ as witnessed by the substitution $\text{id}[\omega_1 \mapsto a, \omega_2 \mapsto a, \omega_3 \mapsto b]$, but $f(a, a, b)$ is only consistent w.r.t. partition P_1 . We can verify that the given term only matches pattern $f(x, x, z)$.

We define a rename function that yields a position annotated term and a consistency partition over $\mathcal{F}(t)$ for any given term.

Definition 4.2. The term rename function $\text{rename} : \mathbb{T}_\Sigma \rightarrow (\mathbb{T}_{\Sigma_{\mathbb{P}}} \times 2^{\mathbb{P}})$ is defined as

$$\text{rename}(t) = (\text{rename}_1(t, \epsilon), \{\{p \in \mathbb{P} \mid t[p] = x\} \mid x \in \text{vars}(t)\})$$

where $\text{rename}_1(t, \epsilon) : (\mathbb{T}_\Sigma \times \mathbb{P}) \rightarrow \mathbb{T}_{\Sigma_{\mathbb{P}}}$ renames the variables of the given term to position annotated variables as follows.

$$\begin{aligned} \text{rename}_1(x, p) &= \omega_p && \text{if } x \in \mathbb{V} \\ \text{rename}_1(f(t_1, \dots, t_n), p) &= f(\text{rename}_1(t_1, p.1), \dots, \text{rename}_1(t_n, p.n)) \end{aligned}$$

Note that for linear patterns, the renaming results in a position annotated term with just trivial consistency partitions that only consist of singleton consistency classes. We show a number of characteristic properties of the rename function which are essential for the correctness of the described two-phase non-linear matching algorithm.

Lemma 4.3. *For all terms $t \in \mathbb{T}_\Sigma$ if $(t', P) = \text{rename}(t)$ then:*

- $t =_\omega t'$, and
- for all $p \in \mathcal{F}(t)$: $t'[p] = \omega_p$, and
- for all $u \in \mathbb{T}_\Sigma$ it holds that u matches t if and only if u matches t' and u is consistent w.r.t. P .

Proof. We can show by induction on t that $t =_\omega \text{rename}_1(t, \epsilon)$ to prove the first statement. For the second statement let $p \in \mathcal{F}(t)$. First, we can show that $t'[p] = \text{rename}_1(t[p], p)$ by induction on position p . From $t[p] \in \mathbb{V}$ it follows that $t'[p]$ is equal to ω_p .

For the last property let P be equal to $\{\{p \in \mathbb{P} \mid t[p] = x\} \mid x \in \text{vars}(t)\}$ and let u be an arbitrary term. Assume that u is consistent w.r.t. P and u matches t' . The latter means that there is a substitution σ such that $t'^\sigma = u$. It follows that for all positions $p \in \mathcal{F}(t')$ that $\sigma(t'[p]) = u[p]$. As u is consistent w.r.t. P it means that for all $x \in \mathbb{V}$ and $p, q \in \mathbb{P}$ that if $t[p] = t[q] = x$ then $u[p] = u[q]$. Therefore, we can construct the substitution ρ such that for all $p \in \mathcal{F}(t)$ we assign $u[p]$ to $t[p]$, where the latter is some variable in $\text{vars}(t)$. The observation of consistency above lets us conclude that there is only one such substitution ρ . From $t =_\omega t'$ it follows that $t^\rho = t'^\sigma$ and as such $t^\rho = u$, which means that u matches t .

Otherwise, if u matches t then there is a substitution σ such that $t^\sigma = u$. Let ρ be the substitution such that for all positions $p \in \mathcal{F}(t)$ we assign $\sigma(t[p])$ (which is equal to $u[p]$) to ω_p . As t' is linear it follows that each ω_p is assigned once and thus $\rho(\omega_p) = \sigma(t[p])$ by definition. Again, from $t =_\omega t'$ it follows that $t'^\rho = t^\sigma$ and as such u matches t' . Finally, for all positions p and q such that $t[p] = t[q] = x$ for variable $x \in \mathbb{V}$ it follows that $u[p] = u[q] = \sigma(x)$. We can thus conclude that u is consistent w.r.t. P . \square

For the variable consistency check a straightforward implementation follows directly from Definition 4.1. Let $P = \{C_1, \dots, C_n\}$ be a consistency partition. For each consistency class C_i , for $1 \leq i \leq n$, there are $|C_i| - 1$ comparisons to perform, after which the consistency of a term w.r.t. C_i is determined. This can be extended to partitions by performing such a check for every consistency class in the given partition. We use the function $\text{IS-CONSISTENT}(t, P)$ to denote this naive algorithm. For a set of partitions $\{P_1, \dots, P_m\}$ the (naive) consistency check requires exactly $\sum_{1 \leq j \leq m} \sum_{C \in P_j} |C| - 1$ comparisons if t is consistent w.r.t. P . Furthermore, it requires at least m comparisons for any term t .

For the renaming procedure we must consider that the patterns $f(x, x)$ and $f(x, y)$ are both renamed to the linear pattern $f(\omega_1, \omega_2)$. However, then it is no longer possible to identify the corresponding original pattern. This can be solved by considering an indexed family of patterns, which is defined as follows. We assume the existence of an index set \mathcal{I} and use $\mathcal{L} \times \mathcal{I}$ to denote the indexed family of patterns with elements denoted by $i : \ell$ for $\ell \in \mathcal{L}$ and $i \in \mathcal{I}$. We adapt the rename function to preserve the index assigned to each pattern. Now, when given an indexed linear pattern that resulted from renaming we can identify the corresponding original pattern by its index. We now combine these results to obtain a non-linear matching algorithm. The correctness of the following lemma follows directly from the third property of Lemma 4.3.

Lemma 4.4. *Let $\mathcal{L} \subseteq \mathbb{T}_\Sigma \times \mathcal{I}$ be an indexed family of patterns and let $\mathcal{L}_r \subseteq \mathbb{T}_{\Sigma^p} \times 2^{2^p} \times \mathcal{I}$ be the indexed family of renamed patterns and corresponding consistency partitions resulting from renaming; i.e., $\mathcal{L}_r = \{\text{rename}(i : \ell) \mid i : \ell \in \mathcal{L}\}$. Let $\text{MATCH-LINEAR} : \mathbb{T}_\Sigma \times 2^{\mathbb{T}_\Sigma} \times \mathcal{I} \rightarrow 2^{\mathbb{T}_\Sigma} \times \mathcal{I}$ be a linear matching function that preserves indices. For any term $t \in \mathbb{T}_\Sigma$ we define $\text{MATCH} : (\mathbb{T}_\Sigma \times (\mathbb{T}_{\Sigma^p} \times 2^{2^p} \times \mathcal{I})) \rightarrow \mathbb{T}_\Sigma$ as:*

$$\text{MATCH}(t, \mathcal{L}_r) = \{\ell \mid i : \ell' \in \mathcal{L}' \wedge i : (\ell', P) \in \mathcal{L}_r \wedge \text{IS-CONSISTENT}(t, P)\}$$

where \mathcal{L}' is equal to $\text{MATCH-LINEAR}(t, \{i : \ell' \mid i : (\ell', P) \in \mathcal{L}_r\})$. The function MATCH is a matching function.

4.2. Consistency Automata. In this section, we are going to focus on solving the consistency checking efficiently by exploiting overlapping partitions to obtain an efficient linear pattern matching algorithm. This is inspired by the APMA where overlapping patterns are exploited to obtain an efficient matching algorithm. Consider the consistency partitions $P_1 = \{\{1, 2\}, \{3\}\}$, $P_2 = \{\{1, 3\}, \{2\}\}$ and $P_3 = \{\{1, 2, 3\}\}$ again. In this case we can use the result of comparisons from overlapping partitions to determine the subset of all consistent partitions efficiently. We would expect that at most three comparisons $t[1] = t[2]$, $t[2] = t[3]$ and $t[1] = t[3]$ would have to be performed to determine the consistent partitions.

To exploit this, we define *consistency automata* which are constructed from a set of consistency partitions. A consistency automaton, abbreviated by CA, is a state machine where every state is either a consistency state, which is labelled with a pair of positions, or a final state, which is labelled with set of partitions. Each consistency state is labelled with a pair of positions that should be compared. Similar labelling is also present in other matching algorithms [Vor95]. The transitions of a CA are labelled with either \checkmark or \times to indicate that the compared positions are equal or unequal respectively. The evaluation of a CA determines the consistency of a term w.r.t. a given set of partitions.

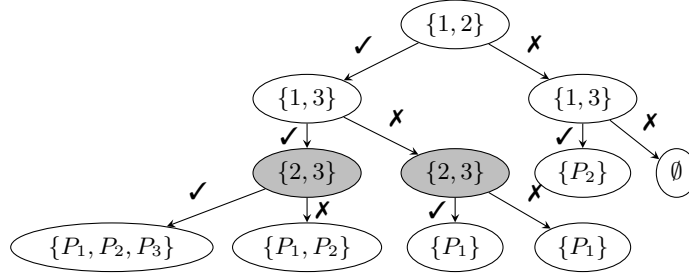


FIGURE 3. The CA for the partitions $P_1 = \{\{1, 2\}, \{3\}\}$, $P_2 = \{\{1, 3\}, \{2\}\}$ and $P_3 = \{\{1, 2, 3\}\}$ where positions 1 and 2 are compared first, followed by 1 and 3 and finally 2 and 3. The grey states are redundant and can be removed as shown later on.

Definition 4.5. A *consistency automaton* is a tuple (S, δ, L, s_0) where:

- $S = S_C \uplus S_F$ is a set of states consisting of a set of consistency states S_C and a set of final states S_F ;
- $\delta : (S_C \times \{\checkmark, \times\}) \rightarrow S$ is a transition function;
- $L = L_C \uplus L_F$ is a state labelling function with $L_C : S_C \rightarrow \mathbb{P}^2$ and $L_F : S_F \rightarrow 2^{\mathcal{I}}$;
- $s_0 \in S$ is the initial state.

We show an example to illustrate the intuition behind the evaluation function of a CA. Consider the consistency partitions $P_1 = \{\{1, 2\}, \{3\}\}$, $P_2 = \{\{1, 3\}, \{2\}\}$ and $P_3 = \{\{1, 2, 3\}\}$ again. Figure 3 shows a CA that can be used to decide the consistency of a given term t w.r.t. any of these partitions. In the state labelled with $\{1, 2\}$, the subterms $t[1]$ and $t[2]$ are compared. Whenever these are equal the evaluation continues with the \checkmark -branch and it continues with the \times -branch otherwise. If a final state (labelled with partitions) is reached then t is consistent w.r.t. these partitions by construction. Afterwards, we show that redundant comparisons can be removed such that this example requires at most two comparisons.

The evaluation function of a CA is defined in Algorithm 3. If the current state is final then the label $L(s)$ indicates the set of indices such that t is consistent w.r.t. the partitions P_i for $i \in L(s)$. Otherwise, evaluation proceeds by considering the pair of positions given by $S_C(s)$. The positions given by $S_C(s)$ are unordered pairs of positions (or 2-sets), denoted by \mathbb{P}^2 , with elements $\{p, q\}$ such that $p \neq q$. These unordered pairs avoid unnecessary comparisons by the reflexivity and symmetry of term equality. If the comparison yields true, the evaluation proceeds with the state of the outgoing \checkmark -transition; otherwise it proceeds with the state of the outgoing \times -transition.

The construction procedure of a CA is defined in Algorithm 4. Its parameters are the automaton M that has been constructed so far, the set of partitions P and the current state s . Additionally, parameter E contains the pairs of positions where the subterms are known to be equal, and similarly N is the set of pairs that are known to be different. Lastly, a selection function `SELECT` is used to define the strategy for choosing the next positions that are compared.

The partitions in P for which a pair $\{p, q\}$ of positions is known to be different are removed as these can not be consistent. The remaining partitions form the set P' . To denote the remaining work concisely we introduce the notation $\subseteq\subseteq$ for the composition of \subseteq and

Algorithm 3 Given the CA $M = (S, \delta, L, s_0)$ and a term $t \in \mathbb{T}_\Sigma$ then $\text{EVAL-CA}(M, t)$ returns the set of indices given by $\text{EVAL-CA}(M, t, s_0)$ such that t is consistent w.r.t. the corresponding partitions used for constructing M .

$$\text{EVAL-CA}(M, t, s) = \begin{cases} L_F(s) & \text{if } s \in S_F \\ \text{EVAL-CA}(M, t, \delta(s, \checkmark)) & \text{if } s \in S_C \wedge t[p] = t[q] \text{ where } \{p, q\} = L_C(s) \\ \text{EVAL-CA}(M, t, \delta(s, \boldsymbol{X})) & \text{if } s \in S_C \wedge t[p] \neq t[q] \text{ where } \{p, q\} = L_C(s) \end{cases}$$

\in ; formally $A \subseteq \in B$ iff $\exists C \in B : A \subseteq C$. Each pair of E that has already been compared is removed from work . The condition on line 4 checks whether there are no choices left to be made. If this is the case then all partitions in P' are consistent by construction and the labelling function is set to yield the partitions P' .

Otherwise, a pair $\{p, q\}$ of positions in work is chosen by the SELECT function and two outgoing transitions are created. A \checkmark -transition is created that is taken during evaluation whenever the subterms at positions p and q are equal and this information is recorded in E . Otherwise, the fact that these are not equal is recorded in N and a corresponding \boldsymbol{X} -transition is created.

Algorithm 4 Given a set of partitions $P = \{P_1, \dots, P_n\}$ and a selection function SELECT then $\text{CONSTRUCT-CA}(P, \text{SELECT})$ computes a CA using $\text{CONSTRUCT-CA}(P, \text{SELECT}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \emptyset, \emptyset)$ that can be used to evaluate the consistent partitions using EVAL-CA .

```

1: procedure  $\text{CONSTRUCT-CA}(P, \text{SELECT}, M, s, E, N)$ 
2:    $P' := \{P_i \in P \mid \neg \exists C \in P_i : \exists \{p, q\} \in N : p, q \in C\}$ 
3:    $\text{work} := \{\{p, q\} \in \mathbb{P}^2 \mid \{p, q\} \subseteq \in P_i \wedge P_i \in P'\} \setminus E$ 
4:   if  $\text{work} = \emptyset$  then
5:      $M := M[S_F := (S_F \cup \{s\}), L_F := L_F[s \mapsto P']]$ 
6:   else
7:      $\{p, q\} := \text{SELECT}(\text{work})$ 
8:      $M := M[S_C := (S_C \cup \{s\}), L_C := L_C[s \mapsto \{p, q\}]]$ 
9:      $M := \text{CONSTRUCT-CA}(P, \text{SELECT}, M[\delta := \delta[(s, \checkmark) \mapsto s']], s', E \cup \{\{p, q\}\}, N)$ 
       where  $s'$  is a fresh unbranded state w.r.t.  $M$ .
10:     $M := \text{CONSTRUCT-CA}(P, \text{SELECT}, M[\delta := \delta[(s, \boldsymbol{X}) \mapsto s']], s', E, N \cup \{\{p, q\}\})$ 
       where  $s'$  is a fresh unbranded state w.r.t.  $M$ .
11:   return  $M$ 

```

The consistency automata obtained from this construction are not optimal, but later on we show that these redundancies can be removed.

4.3. Proof of Correctness. We show the correctness of the construction and evaluation of a CA as defined in Theorem 4.8. In the following statements let $P = \{P_1, \dots, P_n\}$ be a set of partitions where each partition is a finite set of finite consistency classes and let $\text{SELECT} : 2^{\mathbb{P}^2} \rightarrow \mathbb{P}^2$ be any selection function such that $\text{SELECT}(\text{work}) \in \text{work}$ for all non-empty $\text{work} \subseteq 2^{\mathbb{P}^2}$. For the termination of the construction procedure we can show that the number of choices in work strictly decreases at each recursive call.

Lemma 4.6. *The procedure CONSTRUCT-CA(P , SELECT) terminates.*

Proof. Consider the pair of positions $\{p, q\}$ that is taken from `work` at line 7. It is easy to see that $\{p, q\} \notin E$, and $\{p, q\} \notin N$ follows directly from the fact that P' only consists of partitions of which the consistency classes do not contain positions together in a pair of N . Therefore, it follows that in subsequent recursive calls $\{p, q\}$ cannot be in `work` again as either E or N is extended with $\{p, q\}$ and no elements are ever removed from E or N . Furthermore, the execution of all other statements terminates as $\#(P)$ is finite, which also means that $|E|$ and $|N|$ are finite as inserted pairs satisfy $\{p, q\} \subseteq \in P'$. Finally, the selection function terminates by assumption. \square

For the construction procedure we can show that for parameter s it holds that $s \notin S$ as a precondition. Therefore, we can use $\text{work}(s) : S \rightarrow 2^{\mathbb{P}}$, $E(s) : S \rightarrow 2^{\mathbb{P}^2}$ and $N(s) : S \rightarrow 2^{\mathbb{P}^2}$ to denote the values of `work`, E and N respectively during the recursive call of $\text{CONSTRUCT-CA}(P, \text{SELECT}, M, s, E, N)$. For the termination of the evaluation procedure we can show that $\text{work}(s)$ strictly decreases for the visited states.

For the proof of partial correctness we show a relation between the pairs in $E(s)$ and $N(s)$ and the comparisons performed in the evaluation function. First, we define for a term $t \in \mathbb{T}_\Sigma$ and parameters $E, N \subseteq 2^{\mathbb{P}^2}$ the notion of *consistency* where t is consistent w.r.t. E and N , denoted by $(E, N) \models t$, iff:

- $\forall \{p, q\} \in E : t[p] = t[q]$, and
- $\forall \{p, q\} \in N : t[p] \neq t[q]$

A consistency automaton $M = (S, \delta, L, s_0)$ is *well-formed* iff for all terms $t \in \mathbb{T}_\Sigma$ and all recursive calls $\text{EVAL-CA}(M, t, s_0) = \text{EVAL-CA}(M, t, s_n)$ it holds that $(E(s_n), N(s_n)) \models t$.

Lemma 4.7. *Let $M = (S, \delta, L, s_0)$ be the result of CONSTRUCT-CA(P , SELECT). Then M is well-formed.*

Proof. The recursive calls form an *evaluation series* $(s_0, a_0), \dots, (s_n, a_n)$ for $s_i \in S$ and $a_i \in \{\checkmark, \times\}$ for $0 \leq i < n$ such that $\text{EVAL-CA}(M, s_i, t) = \text{EVAL-CA}(M, s_{i+1}, t)$ and $\delta(s_i, a_i) = s_{i+1}$. Let $t \in \mathbb{T}_\Sigma$ be any term. We prove the statement by induction on the length of the evaluation series.

Base case. We have $E(s_0) = N(s_0) = \emptyset$ and as such the statement holds vacuously.

Inductive step. Suppose that for $\text{EVAL-CA}(M, t, s_0) = \text{EVAL-CA}(M, t, s)$ the statement holds. Suppose that $\text{EVAL-CA}(M, t, s) = \text{EVAL-CA}(M, t, s')$ where $s' = \delta(s, a)$ for $a \in \{\checkmark, \times\}$ and let $L_C(s) = \{p, q\}$. There are two cases to consider:

- $t[p] = t[q]$ in which case $E(s')$, where s' is equal to $\delta(s, \checkmark)$, is $E(s)$ extended with $\{p, q\}$ and $N(s') = N(s)$.
- Otherwise, $t[p] \neq t[q]$ in which case $N(s')$ is equal to $N(s)$ extended with $\{p, q\}$ and $E(s') = E(s)$.

In both cases $(E(s'), N(s')) \models t$ holds by definition. \square

Finally, we can show the correctness of using consistency automata to evaluate the consistency of a given term w.r.t. partitions in P .

Theorem 4.8. *Let $M = (S, \delta, L, s_0)$ be the result of CONSTRUCT-CA(P , SELECT). Consider an arbitrary term t and suppose that $\text{EVAL-CA}(M, s_0, t) = P'$. Then for all $P_j \in P$ it holds that $P_j \in P'$ iff the term t is consistent w.r.t. P_j .*

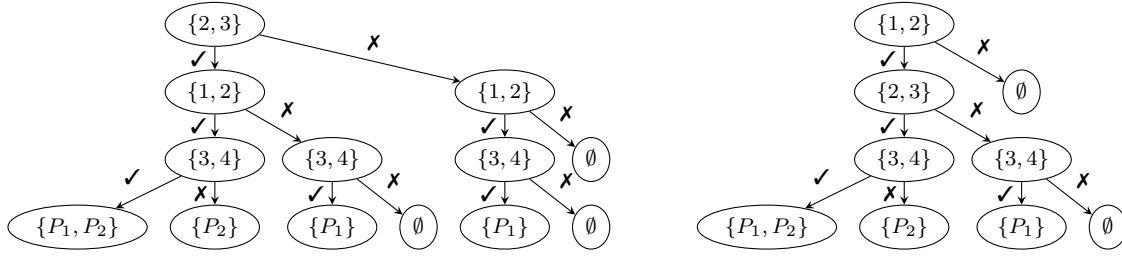


FIGURE 4. Two CAs for the partitions $P_1 = \{\{1, 2\}, \{3, 4\}\}$ and $P_2 = \{\{1, 2, 3\}\}$. The CA on the left chooses $\{2, 3\}$ first. However, as shown on the right selecting $\{1, 2\}$ first removes both partitions, and leads to a smaller CA.

Proof. We have already shown termination of the construction procedure in Lemma 4.6. Let P' be the set of partitions returned by $\text{EVAL-CA}(M, t, s_0)$, let $P_i \in P$ be any partition and $\text{EVAL-CA}(M, t, s_0) = \text{EVAL-CA}(M, t, s_n)$ for some final state $s_n \in S_F$. By Lemma 4.7 it holds for all $\{p, q\} \in E(s_n)$ that $t[p] = t[q]$ and for all $\{p, q\} \in N(s_n)$ that $t[p] \neq t[q]$.

\implies) Assume that $P_j \in P'$. For all p, q such that $\{p, q\} \subseteq P_j$ it holds that $\{p, q\} \in E(s_n)$ as $\text{work}(s_n)$ is equal to \emptyset for s_n to become a final state in the construction. Therefore, for all $p, q \in C$ for consistency class $C \in P_j$ it holds that $t[p] = t[q]$ and as such t is consistent w.r.t. P_j .

\impliedby) Assume that term t is consistent w.r.t. P_j . Proof by contradiction, assume that $P_j \notin P'$. As such, there is a position pair $\{p, q\} \subseteq P_j$ such that $\{p, q\} \in N(s_n)$. However, then it follows that $t[p] \neq t[q]$, from which we conclude that t can not be consistent w.r.t. P_j . \square

4.4. Efficiency. Similarly to APMAs we can consider the space and time measurements for CAs. Given a CA $M = (S, \delta, L, s_0)$ and a term t we define the *evaluation depth*, denoted by $\text{ED}(M, t)$, as the number of recursive EVAL-CA calls performed to reach the final state. The size, denoted by $|M|$, is given by the number of states $|S|$. Note that here we refer to the total number of states for its size, instead of only the number of final states or equivalently its breadth. We also define the notion of relative efficiency for CAs.

Definition 4.9. Given two consistency automata $M = (S, \delta, L, s_0)$ and $M' = (S', \delta', L', s'_0)$ for a set of consistency partitions P . We say that $M \preceq M'$ iff for all terms $t \in \mathbb{T}_\Sigma$ it holds that $\text{ED}(M, t) \leq \text{ED}(M', t)$.

We present two ways to improve the time and space efficiency of consistency automata. First of all, the selection function used for construction influences the size and the evaluation time for the resulting CA as shown in Figure 4. Choosing the pair of positions $\{2, 3\}$ before $\{1, 2\}$ in this example yields a CA that is both larger in size and less efficient. Therefore, it makes sense to consider heuristics for the selection function in practice. For instance, taking the selection function that always picks positions $\{p, q\}$ to minimise the size of work in both branches would immediately yield the right CA in Figure 4. In cases where there is no overlap between the partitions the choice would be arbitrary.

Changing the selection function by itself does not necessarily result in the most relatively efficient CA. If we consider Figure 3 again, we can observe that the resulting automaton

is not optimal, despite being the smallest w.r.t. the selection function, because some of the final states are not reached during evaluation of any term. For example, the final state labelled with $\{P_1, P_2\}$ is not reachable, because any term $t \in \mathbb{T}_\Sigma$ that satisfies $t[1] = t[2]$ and $t[1] = t[3]$ can not have that $t[2] \neq t[3]$ by the transitivity of term equality. Therefore, we can consider removing these states such that the evaluation depth of terms where evaluation reaches these states is lower. This is essentially the notion of redundancies that was previously defined for APMAAs.

Given a CA $M = (S, \delta, L, s_0)$ and a non-final state $s \in S_C$ we give the following conditions for its redundancy.

Definition 4.10 (Redundancy for CAs). Let $M = (S, \delta, L, s_0)$ be a CA.

- A consistency state s is \checkmark -redundant iff for all terms t , whenever $\text{EVAL-CA}(M, t, s_0) = \text{EVAL-CA}(M, t, s)$, then $\text{EVAL-CA}(M, t, s) = \text{EVAL-CA}(M, t, \delta(s, \checkmark))$.
- A consistency state s is \times -redundant iff for all terms t , whenever $\text{EVAL-CA}(M, t, s_0) = \text{EVAL-CA}(M, t, s)$, then $\text{EVAL-CA}(M, t, s) = \text{EVAL-CA}(M, t, \delta(s, \times))$.

The notion of *dead* states does not apply for CAs, because we explicitly keep final states labelled with the empty set for easier definitions.

Redundant states can be removed from the automata without affecting the correctness of its evaluation in the following way. A state s that is \checkmark -redundant can be *removed* by updating δ such that the incoming transition $\delta(r, a) = s$, for some $r \in S$ and $a \in \{\checkmark, \times\}$, is updated to $\delta(s, \checkmark)$. A similar transformation of δ can be applied for states that are \times -redundant using $\delta(s, \times)$. We can observe that such a removal reduces the evaluation depth for terms where evaluation reached this state by one and that the size of the CA is reduced by the number of states in the \checkmark -branch (or \times -branch) respectively if states unreachable by the transition relation are removed. Next, we prove that removal does not influence the correctness of evaluation.

Lemma 4.11. *Let $M = (S, \delta, L, s_0)$ be any CA that is well-formed. Then the resulting CA M' where a \checkmark -redundant or \times -redundant state $v \in S$ is removed remains well-formed.*

Proof. The recursive calls form an *evaluation series* $(s_0, a_0), \dots, (s_n, a_n)$ for $s_i \in S$ and $a_i \in \{\checkmark, \times\}$ for $0 \leq i < n$ such that $\text{EVAL-CA}(M, s_i, t) = \text{EVAL-CA}(M, s_{i+1}, t)$ and $\delta(s_i, a_i) = s_{i+1}$. By well-formedness of M we know, for all terms $t \in \mathbb{T}_\Sigma$ and all evaluation series $(s_0, a_0), \dots, (s_k, a_k) \in (S \times \{\checkmark, \times\})$ of $\text{EVAL-CA}(M, s_0, t)$, that for all states s_i , with $0 \leq i \leq k$, it holds that $(E(s_i), N(s_i)) \models t$. Now, we only have to consider sequences that contain the state v as the other evaluation sequences remain the same. Consider any such sequence and let u be the state in that sequence such that $\delta(u, a) = v$, for some $a \in \{\checkmark, \times\}$, and let t be an arbitrary term. Let $\{p, q\}$ be the value of $L_C(v)$ then there are two cases to consider:

- v is \checkmark -redundant. It follows that $t[p] = t[q]$ for $\{p, q\} := L_C(v)$. All sequences such that v occurs in it must contain exactly the pair (v, \checkmark) by definition of \checkmark -redundancy. We conclude that $(E(u) \cup \{\{p, q\}\}, N(u)) \models t$ holds and the term remains consistent with all extensions to E and N for the remaining states in the sequence.
- v is \times -redundant. Similarly, with the observation that $(E(u), N(u) \cup \{\{p, q\}\}) \models t$. \square

Using Lemma 4.11 and the fact that removing redundant states does not change the labelling of any state we have shown that $\text{EVAL-CA}(M, s_0, t) = \text{EVAL-CA}(M', s_0, t)$ for all t . Instead of removing redundant states after constructing the CA we could also remove them on-the-fly during the construction. Whenever a pair of positions $\{p, q\}$ *would* result in a

✓-redundant (or ✗-redundant) state we could instead change the parameters N and E to avoid such a choice. Namely, whenever choice $\{p, q\}$ would result in a ✓-redundant state then we could update E to become $E \cup \{\{p, q\}\}$, and similarly for parameter N in case of a ✗-redundant state. This means that the construction essentially continues as if this choice had already been made, thus avoiding the creation of redundant states.

If we consider Figure 3 again it follows from transitivity that the left indicated state is ✓-redundant and the right indicated state ✗-redundant. If the indicated states are removed then all states of the resulting CA are reachable, which could be argued for as a form of optimum. For transitivity it is relatively straightforward to construct a procedure to identify and remove these states. However, it would be more interesting to devise a method that determines all redundant states. For example, there can also be redundancies due to the fact that a term can never be equal to any of its subterms. Later on, we see that even more redundancies can be observed from the interleaving of matching and consistency states.

4.5. Time and Space Complexity. For the time complexity we consider the number of comparisons performed to determine the consistency. Given a set of partitions $P = \{P_1, \dots, P_n\}$ we have already established a worst-case time complexity for naive consistency checking as being $\mathcal{O}(\sum_{1 \leq j \leq n} \sum_{C \in P_j} |C| - 1)$ with a worst-case space complexity of $\mathcal{O}(1)$.

We establish several upper and lower bounds on the space and time complexity for CAs. The number of comparisons performed for a given term is determined by the evaluation depth, which is the measurement for time complexity. The upper bound is then given by $\max_{t \in \mathbb{T}_\Sigma} (\text{ED}(M, t))$. Similarly, we could define the minimal evaluation depth as the lower bound. However, since we include final states labelled with the empty set in the CA this would simply be 1. Therefore, we consider the lower bound to be the minimum evaluation depth reaching any final state labelled with at least one consistency partition. The selection function is a parameter and thus we need to consider what CA will be used for the complexity analysis. For the upper bound on time we consider the automaton where the bound is minimised w.r.t. all possible selection function, and similarly for the upper bound on space. On the other hand, the lower bound for both the time and space is for *any* selection function.

For the upper bound of time complexity of consistency automata we can show that each pair of positions is compared at most once. Let m be the number of unique position pairs in the given partitions, where each pair of positions is counted at most once. First, we show that the worst-case time complexity of the consistency automata evaluation is tightly bounded by $\mathcal{O}(m)$. This is essentially the size of *work* for the first call to the construction procedure, which reduces in each recursive call.

Lemma 4.12. *Let $P = \{P_1, \dots, P_n\}$ be a set of partitions and let M be the optimal CA resulting from $\text{CONSTRUCT-CA}(P, \text{SELECT})$ for some selection function SELECT . For any term $t \in \mathbb{T}_\Sigma$ the evaluation depth $\text{ED}(M, t)$ is of complexity $\mathcal{O}(m)$, where m is the number of unique position pairs in P .*

Proof. Initially $\text{work} = \{\{p, q\} \subseteq P'\}$ contains at most m pairs by assumption. As shown in the proof of Lemma 4.6 each choice of *work* can be made at most once by SELECT . Therefore, the length of the root to a final state of the resulting CA is at most $\mathcal{O}(m)$, which is also an upper bound for number of comparisons. \square

Furthermore, each recursive call leads to exactly two branches, which means that the size of any CA is bounded by $\mathcal{O}(2^m)$. The given bounds are also tight as we can construct the following example where the evaluation depth requires exactly m comparisons, which in this example coincides with the number of comparisons in the naive approach.

Example 4.13. Let P be a set of partitions $\{\{p, q\} \mid p, q \in \{1, \dots, k\}\}$, which contains exactly $\frac{k(k-1)}{2}$ consistency classes. In each recursive call $|\text{work}|$ decreases by exactly one, which means that it takes $\frac{k(k-1)}{2}$ comparisons to yield the set of partitions in the worst case. Then the automaton without removing redundant states contains $2^{\frac{k(k-1)}{2}}$ states.

Therefore, it follows that the upper bound on time complexity of the consistency automata is tightly bounded up to k^2 position pairs, where k is the number of positions. Any additional partition, which necessarily only contains pairs of positions that have already been compared, can be inferred from the comparisons that have already been made. Note that the upper bound on evaluation depth in this example cannot be reduced by removing redundant states, because every state on the path consisting of only \mathbf{X} -transitions does not contain redundant states. Namely, it is fairly straightforward to show that there is a term $f(t_0, \dots, t_n)$ where the subterms at two positions are equal and the subterms at any *other* pair of positions are unequal. Therefore, at any chain of \mathbf{X} -transitions it is still necessary to check whether the \checkmark -transition should be taken and as such the maximum evaluation depth also remains strict for CAs where redundant states have been removed.

The lower bound on evaluation depth for CAs where redundant states have been removed is given by number of positions in the smallest consistency partition. This bound is also strict, because we can simply give an example with a single consistency partition. Removing redundant states by taking transitivity into account is sufficient to ensure that the evaluation depth of the consistency automata never exceeds the worst-case time required for the naive consistency check. However, the space required for using CAs is always higher than the naive consistency check.

5. ADAPTIVE NON-LINEAR PATTERN MATCHING AUTOMATA

We have shown in Lemma 4.4 that a naive matching algorithm for non-linear patterns can be obtained by using a linear matching function followed by a consistency check. In that case we have to check the consistency of all partitions returned by the linear matching function. However, as shown in the following example overlapping patterns can unify with the same prefix, but no term can match both patterns at the same time.

Consider the patterns: $\ell_1 : f(x, x)$ and $\ell_2 : f(a, b)$. After renaming we obtain the following pairs $\ell_1 : (f(\omega_1, \omega_2), \{\{1, 2\}\})$ and $\ell_2 : (f(a, b), \{\emptyset\})$. Now, the resulting APMA has a final state labelled with both patterns as shown in Figure 5a. We can observe that the consistency check of positions one and two always yields false whenever the evaluation of a term ends up in the final state labelled with $\{\ell_1, \ell_2\}$, because terms a and b are not equal. Therefore, this comparison would be unnecessary.

We could also consider an alternative where the consistency phase is performed first, but then we have the problem that whenever the given term is consistent w.r.t. partition $\{1, 2\}$ that matching on $f(a, b)$ should be avoided. To avoid such redundancies, we propose a combination of APMA and CA to obtain a matching automaton for non-linear patterns called *adaptive non-linear pattern matching automata*, abbreviated as ANPMAs. The result

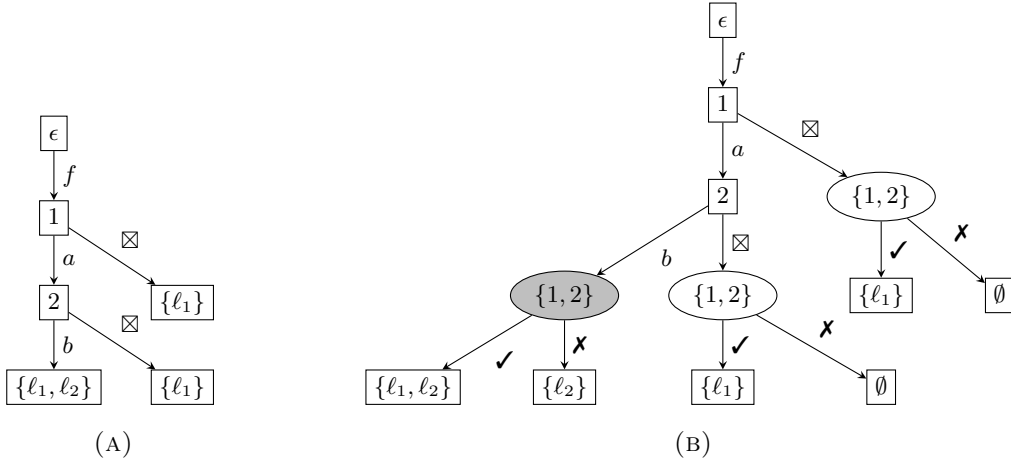


FIGURE 5. The resulting APMA shown on the left and the corresponding ANPMA with a grey \mathbf{X} -redundant state on the right.

is an automaton that has three kinds of states; match states of APMAs, consistency states of CAs and final states, and two transition functions; one for match states and one for consistency states.

Definition 5.1. An adaptive non-linear pattern matching automaton (ANPMA) is a tuple (S, δ, L, s_0) with

- $S = S_M \uplus S_C \uplus S_F$ is a set of states where S_M is a set of match states, S_C is a set of consistency states and S_F is a set of final states;
- $\delta = \delta_F \uplus \delta_C$ is a partial transition function with $\delta_F : S_M \times \mathbb{F} \rightarrow S$ and $\delta_C : S_C \times \{\checkmark, \mathbf{X}\} \rightarrow S$;
- $L = L_M \uplus L_C \uplus L_F$ is a state labelling function with $L_M : S_M \rightarrow \mathbb{P}$, $L_C : S_C \rightarrow \mathbb{P}^2$ and $L_F : S_F \rightarrow 2^{\mathbb{T}}$;
- $s_0 \in S_M$ is the initial state.

We only consider ANPMAs that have a tree structure rooted in s_0 . Given an ANPMA $M = (S, \delta, L, s_0)$ and a term t the procedure $\text{MATCH}(M, s_0, t)$ defined in Algorithm 5 defines the evaluation of the ANPMA. It is essentially the combination of the evaluation functions for the APMAs and CAs depending on the current state.

Algorithm 5 Given a state s of the ANPMA $M = (S, \delta, L, s_0)$ and a term t , the following algorithm computes the pattern matches of t by evaluating M on t .

$$\text{MATCH}(M, t, s) = \begin{cases} L_M(s) & \text{if } s \in S_F \\ \text{MATCH}(M, t, \delta_F(s, f)) & \text{if } s \in S_M \wedge \delta(s, f) \neq \perp \\ \text{MATCH}(M, t, \delta_F(s, \boxtimes)) & \text{if } s \in S_M \wedge \delta(s, \boxtimes) \neq \perp \wedge \delta(s, f) = \perp \\ \emptyset & \text{if } s \in S_M \wedge \delta(s, \boxtimes) = \delta(s, f) = \perp \\ \text{MATCH}(M, t, \delta_C(s, \checkmark)) & \text{if } s \in S_C \wedge t[p] = t[q] \\ \text{MATCH}(M, t, \delta_C(s, \mathbf{X})) & \text{if } s \in S_C \wedge t[p] \neq t[q] \end{cases}$$

where $f = \text{head}(t[L_M(s)])$ and $\{p, q\} = L_C(s)$

The construction algorithm of the ANPMA is defined in Algorithm 6. It combines the construction algorithm of APMAs (Algorithm 2) and the construction algorithm for CAs

(Algorithm 4). The parameters that remain the same value during the recursion are the original set \mathcal{L} , the result of renaming \mathcal{L}_r and the selection function SELECT. Next, we have the ANPMA M , a state s and finally the current prefix pref similar to the APMA construction and the sets of position pairs E and N as in the consistency automata construction.

Algorithm 6 Given an indexed family of patterns $\mathcal{L} \subseteq \mathbb{T}_\Sigma \times \mathcal{I}$ and an indexed family of renamed patterns $\mathcal{L}_r \subseteq \mathbb{T}_{\Sigma_p} \times 2^{\mathbb{P}} \times \mathcal{I}$ this algorithm computes an ANPMA for \mathcal{L} . Initially it is called with $M = (\emptyset, \emptyset, \emptyset, s_0)$, the initial state $s = s_0$, the prefix $\text{pref} = \omega_\epsilon$, and $E = N = \emptyset$.

```

1: procedure CONSTRUCTANPMA( $\mathcal{L}, \mathcal{L}_r, \text{SELECT}, M, s, \text{pref}, E, N$ )
2:    $\mathcal{L}'_r := \{i : (\ell, P) \in \mathcal{L}_r \mid \ell \text{ unifies with } \text{pref} \wedge \neg \exists C \in P : \exists \{p, q\} \in N : p, q \in C\}$ 
3:    $\text{workF} := \mathcal{F}(\text{pref})$ 
4:    $\text{workC} := \{\{p, q\} \in \mathbb{P}^2 \mid \{p, q\} \subseteq P_i \wedge (i : \ell, i : P_i) \in \mathcal{L}'_r \wedge \text{pref}[p] \text{ and } \text{pref}[q] \text{ are defined}\} \setminus E$ 
5:   if ( $\text{workF} = \emptyset$  and  $\text{workC} = \emptyset$ ) or ( $\mathcal{L}'_r = \emptyset$ ) then
6:      $M := M[S_F := S_F \cup \{s\}, L := L[s \mapsto \{i : \ell \in \mathcal{L} \mid \exists \ell', P : i : (\ell', P) \in \mathcal{L}'_r\}]]$ 
7:   else
8:      $\text{next} := \text{SELECT}(\text{workF}, \text{workC})$ 
9:     if  $\text{next} = \text{pos}$  for some position  $\text{pos}$  then
10:       $M := M[S_M := (S_M \cup \{s\}), L_M := L_M[s \mapsto \text{pos}]]$ 
11:       $F := \{f \in \mathbb{F} \mid \exists (i : (\ell, P)) \in \mathcal{L}'_r : \text{head}(\ell[\text{pos}]) = f\}$ 
12:      for  $f \in F$  do
13:         $M := M[\delta := \delta[(s, f) \mapsto s']]$  where  $s'$  is a fresh unbranded state w.r.t.  $M$ 
14:         $M := \text{CONSTRUCTANPMA}(\mathcal{L}, \mathcal{L}_r, \text{SELECT}, M, s', \text{pref}[\text{pos}/f(\omega_{\text{pos}.1}, \dots, \omega_{\text{pos.ar}(f)})], E, N)$ 
15:      if  $\exists (i : (\ell, P)) \in \mathcal{L}'_r : \exists \text{pos}' \leq \text{pos} : \text{head}(\ell[\text{pos}']) \in \mathbb{V}$  then
16:         $M := M[\delta := \delta[(s, \boxtimes) \mapsto s']]$  where  $s'$  is a fresh unbranded state w.r.t.  $M$ 
17:         $M := \text{CONSTRUCTANPMA}(\mathcal{L}, \mathcal{L}_r, \text{SELECT}, M, s', \text{pref}[\text{pos}/\boxtimes], E, N)$ 
18:      else if  $\text{next} = \{p, q\}$  for some pair  $\{p, q\} \in \mathbb{P}^2$  then
19:         $M := M[S_C := (S_C \cup \{s\}), L_C := L_C[s \mapsto \{p, q\}]]$ 
20:         $M := \text{CONSTRUCTANPMA}(\mathcal{L}, \mathcal{L}_r, \text{SELECT}, M[\delta_C := \delta_C[(s, \checkmark) \mapsto s']], s', \text{pref}, E \cup \{\{p, q\}\}, N)$ 
        where  $s'$  is an unbranded state w.r.t.  $M$ .
21:         $M := \text{CONSTRUCTANPMA}(\mathcal{L}, \mathcal{L}_r, \text{SELECT}, M[\delta_C := \delta_C[(s, \boldsymbol{X}) \mapsto s']], s', \text{pref}, E, N \cup \{\{p, q\}\})$ 
        where  $s'$  is an unbranded state w.r.t.  $M$ .
22:   return  $M$ 

```

First we remove the terms that do not have to be considered anymore. These are the elements $i : (\ell, P)$ from \mathcal{L}_r such that P is inconsistent due to the pairs in N and pref does not unify with ℓ . Obtaining work for both types of choices is almost the same as before. However, for workC we have added the condition that the positions must be defined in the prefix to ensure that these positions are indeed defined when evaluating a term. The termination condition is that both workF and workC are empty, or that the set of patterns \mathcal{L}'_r has become empty. The latter can happen when the inconsistency of two positions removes a pattern, which could still have other positions to be matched.

The function SELECT is a function that chooses a position from workF or a pair of positions from workC . Its result determines the kind of state that s becomes and as such also the outgoing transitions. If a position is selected then s will become a match state and the construction continues as in Algorithm 2. Otherwise, similar to Algorithm 4 two fresh states and two outgoing transition labelled with \checkmark and \boldsymbol{X} are created, after which the parameters E and N are updated.

At this point we can also see that ANPMAs are a natural extension of APMA and CAs. In the worst case we can first select only choices in workF and whenever workF is empty we construct only consistency states. This would be exactly the same as performing

a linear matching algorithm using APMA's followed by a consistency check using CAs, which is exactly the two-phase approach.

5.1. Correctness. The ANPMA construction algorithm yields an ANPMA that is suitable to solve the matching problem for non-empty finite sets of (non-linear) patterns. This can be shown by combining the efforts of Theorem 3.8 and Theorem 4.8.

Let \mathcal{L} be a finite non-empty indexed family of (non-linear) patterns and let $(\mathcal{L}_r, P) = \text{rename}(\mathcal{L})$. Suppose that $\text{SELECT} : 2^{\mathbb{P}} \times 2^{\mathbb{P}^2} \rightarrow \mathbb{P} \uplus \mathbb{P}^2$ is any function such that for all sets of positions workF and position pairs workC we have that $\text{SELECT}(\text{workF}, \text{workC}) \in \text{workF} \uplus \text{workC}$.

We extend the auxiliary definitions for APMA's as follows. A *path* to s_n is a sequence with both types of labels $(s_0, a_0), \dots, (s_{n-1}, a_{n-1}) \in S \times (\mathbb{F}_{\boxtimes} \uplus \{\checkmark, \mathbf{X}\})$ such that $\delta(s_i, a_i) = s_{i+1}$ for all $i < n$. A position p is called *visible* for state s iff there is a pair (s_i, a_i) in $\text{path}(s)$ such that $L(s_i).i = p$ for some $1 \leq i \leq \text{ar}(f_i)$ or $L(s) = \epsilon$. A state s is *top-down* iff $s \in S_M$ and $L_M(s)$ is visible or $s \in S_C$ and both positions in $L_C(s)$ are visible. State s is *canonical* iff there are no two match states in $\text{path}(s)$ that are labelled with the same position. Finally we say that an ANPMA is *well-formed* iff $L(s_0) = \epsilon$, and all states are top-down and canonical.

Lemma 5.2. *The procedure $\text{CONSTRUCTANPMA}(\mathcal{L}_r, P, \text{SELECT}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon, \emptyset, \emptyset)$ terminates and yields a well-formed ANPMA.*

Proof. We only show that the recursion terminates. The rest is similar to the proof for Lemma 3.3, with the additional observation that positions in P are only chosen when they are defined in the prefix. Given the parameters pref_1, E_1, N_1 and pref_2, E_2, N_2 we can fix the ordering:

$$\begin{aligned} & (\text{pref}_1 < \text{pref}_2 \wedge E_1 = E_2 \wedge N_1 = N_2) \vee \\ & (\text{pref}_1 = \text{pref}_2 \wedge E_1 \subset E_2 \wedge N_1 = N_2) \vee \\ & (\text{pref}_1 = \text{pref}_2 \wedge E_1 = E_2 \wedge N_1 \subset N_2). \end{aligned}$$

The prefixes are again only defined on positions that are defined in patterns of \mathcal{L} and the sets E and N are bounded by a finite product of positions, hence the ordering is well-founded. The recursive calls conform to this ordering; therefore the recursion terminates. \square

Let $M = (S, \delta, L, s_0)$ be the ANPMA resulting from $\text{CONSTRUCTANPMA}(\mathcal{L}, \text{SELECT})$. Let $t \in \mathbb{T}_\Sigma$ be a term and \mathcal{L}_t be equal to $\{i : \ell \in \mathcal{L} \mid \ell \leq t\}$. For every state $s \in S$ we define $\mathcal{L}(s)$ to be equal to $\{i : \ell \in \mathcal{L} \mid i : (\ell', P) \in \mathcal{L}'_r(s_i)\}$. We show that the evaluation algorithm on M satisfies a number of invariants.

Lemma 5.3. *For all $s \in S$ such that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$ it holds that: (a) $(E(s_i), N(s_i)) \models t$, (b) $\mathcal{L}_t \subseteq \mathcal{L}(s)$ and (c) if $s \in S_F$ then $L(s_f) = \mathcal{L}_t$.*

Proof. Take an arbitrary term t . We prove the first two invariants by induction on the length of $\text{path}(s)$.

Base case, the empty path and as such $s = s_0$. $E(s_0) = N(s_0) = \emptyset$ and $\mathcal{L}_t \subseteq \mathcal{L}$, and $\mathcal{L} = \mathcal{L}(s_0) = \mathcal{L}(s)$, as such the statements hold vacuously.

Inductive step. Let s be an arbitrary state and suppose that the statements hold for $\text{MATCH}(A, t, s_0) = \text{MATCH}(A, t, s)$. Suppose $\text{MATCH}(A, t, s) = \text{MATCH}(A, t, s')$ for some $s' = \delta(s, x)$ such that $x \in (\mathbb{F}_{\boxtimes} \uplus \{\checkmark, \mathbf{X}\})$. Now, there are two cases to consider:

- $s \in S_C$. Let $\{p, q\}$ be the value of $L_C(s_k)$. Again, there are two cases to consider:

- $t[p] = t[q]$ in which case $E(s')$ is $E(s) \cup \{p, q\}$ and $N(s') = N(s)$. Therefore, $(E(s'), N(s')) \models t$ holds. Furthermore, $\mathcal{L}(s') = \mathcal{L}(s)$ because also $\text{pref}(s') = \text{pref}(s)$.
- Otherwise, $t[p] \neq t[q]$ in which case $N(s')$ is equal to $N(s) \cup \{p, q\}$ and $E(s') = E(s)$. Therefore, $(E(s'), N(s')) \models t$ holds. Consider any $i : \ell \in \mathcal{L}(s)$ such that $i : \ell \notin \mathcal{L}(s')$. From $\text{pref}(s') = \text{pref}(s)$ it follows that for $i : (\ell', P) \in \mathcal{L}_r$ it holds that P is not consistent w.r.t. t by observation that positions $\{p, q\} \subseteq P$ are included in N and $t[p] \neq t[q]$. Therefore, by Lemma 4.3 it holds that $i : \ell \notin \mathcal{L}_t$.
- $s \in S_M$. It holds that $E(s') = E(s)$ and $N(s') = N(s)$. Therefore, $(E(s'), N(s')) \models t$ remains true. Now, we can use the same argument as before to argue that any pattern removed must not unify with $\text{pref}(s')$. Then the same arguments as given in Lemma 3.5 can be used to show that $\mathcal{L}_t \subseteq \mathcal{L}(s')$ holds.

Finally, if $s \in S_F$ from the fact that $L(s) = \mathcal{L}(s)$ we know that $\mathcal{L}_t \subseteq L(s)$. It only remains show that $L(s_f) \subseteq \mathcal{L}_t$. There are two cases for this state to become a final state during construction:

- Both $\text{workC} = \emptyset$ and $\text{workF} = \emptyset$. Suppose for a contradiction that there is some $i : \ell \in L(s_f)$ such that $i : \ell \notin \mathcal{L}_t$. It follows that $\ell \not\leq t$, which means that for $i : (\ell', P) \in \mathcal{L}_r$ that $\ell' \not\leq t$ or t is not consistent w.r.t. P by Lemma 4.3. We show that both cases lead to a contradiction:
 - Case $\ell' \not\leq t$. This follows essentially from the same observations as Lemma 3.7.
 - Case t is not consistent w.r.t. P . From the fact that $\text{pref}(s_f)$ unifies with t and that it is a ground term due to $\text{workF} = \emptyset$ it follows that for all p, q such that $\{p, q\} \subseteq P_i$ they are defined in $\text{pref}(s)$ and therefore it holds that $\{p, q\} \in E(s)$. Therefore, for all $p, q \in C$ for consistency class $C \in P_i$ it holds that $t[p] = t[q]$ and as such t is consistent w.r.t. P_i . As such i is not an element of $L(s_f)$, contradicting our assumption.
- The set $\mathcal{L}(s)$ is empty. In this case $L(s_f)$ is empty and $L(s_f) \subseteq \mathcal{L}_t$ by definition. \square

Lemma 5.4. *If $\mathcal{L}_t = \emptyset$ then $\text{MATCH}(M, t, s_0) = \emptyset$.*

Proof. We show that $\text{MATCH}(M, t, s_0) \neq L(s)$ for all final states s for which $L(s) \neq \emptyset$. Let s_f be an arbitrary final state such that $L(s_f) \neq \emptyset$ and pick some pattern $i : \ell \in L(s_f)$. By assumption $\ell \not\leq t$ and by Lemma 4.3 it holds for the pair $i : (\ell', P) \in \mathcal{L}_r$ that $\ell' \not\leq t$ or t is not consistent w.r.t. P .

- If $\ell' \not\leq t$ then by Proposition 3.1 it follows that there is a position p and a function symbol $f \in \mathbb{F}$ such that $\text{head}(\ell[p]) = f$ and $\text{head}(t[p]) \neq f$. By Lemma 3.4 it must be that $\text{head}(\text{pref}(s)[p]) = f$, by which there must be a pair $(s_i, f) \in \text{path}(s)$. Since MATCH is a function we again have that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_i) = \text{MATCH}(M, t, s_f)$. However, by its definition we know that $\text{head}(t[p]) = f$, which contradicts the assumption that $i : \ell \in L(s_f)$.
- If t is not consistent w.r.t. P . By Lemma 5.3 we know that $(E(s_f), N(s_f)) \models t$ and for all pairs $\{p, q\} \subseteq P$ it holds that $\{p, q\} \in E$ for workC to become empty, because all positions of pattern ℓ are defined in the prefix $\text{pref}(s_f)$. As such t must be consistent w.r.t. P , which contradicts the assumption that $i : \ell \in L(s_f)$. \square

Theorem 5.5. *Then $\lambda t. \text{MATCH}(M, t, s_0)$ is a matching function for \mathcal{L} .*

Proof. If \mathcal{L}_t is empty then by Lemma 5.4 we get that $\text{MATCH}(M, t, s_0) = \emptyset = \mathcal{L}_t$ as required. Otherwise, we have that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_f)$ for some final state s_f . Then by the definition of MATCH and Lemma 5.3 we conclude $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_f) = L(s_f) = \mathcal{L}_t$. \square

5.2. The redundancy problem for ANPMAs. We have seen the notion of f -redundant states and dead states for APMAs in Section 3. In particular Lemma 3.10 shows that the construction by Sekar et al. eliminates all such redundancies. We have also discussed the notion of \checkmark -redundancy and \times -redundancy in Section 4. We repeat the notions here.

Definition 5.6. We observe the following redundancies for ANPMAs.

- Given $f \in \mathbb{F}_{\boxtimes}$, we say that a consistency state s is f -redundant iff for all terms t , whenever $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$, then $\text{MATCH}(M, t, s) = \text{MATCH}(M, t, \delta(s, f))$.
- A consistency state s is \checkmark -redundant iff, for all terms t , whenever $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$, then $\text{MATCH}(M, t, s) = \text{MATCH}(M, t, \delta(s, \checkmark))$.
- A consistency state s is \times -redundant iff, for all terms t , whenever $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$, then $\text{MATCH}(M, t, s) = \text{MATCH}(M, t, \delta(s, \times))$.
- We say that a state s is *dead* iff for all terms t , whenever $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$, then $\text{MATCH}(M, t, s) = \emptyset$.

There is a connection between the pairs in E and N , and the observed function symbols that are represented by `pref`. For example, if one has (partial) knowledge of the function symbols in t_1 and one also knows that $t_1 = t_2$, then the same knowledge of the function symbols of t_2 follows by definition of term equality. The other way around we have that full knowledge of the function symbols of both t_1 and t_2 makes the equality check between them redundant. And even though checking the functions symbols of t_1 first and then comparing t_1 with t_2 to determine a match, we also need to be able to do this the other way around for completeness of the optimisation.

The removal of redundant states in ANPMAs is a difficult problem in its full generality. We describe some observations and some examples in the remainder of this section, but we leave the full problem open for future work.

Suppose that we have a recursive call in the construction with the parameters \mathcal{L}_r , `pref`, E and N . Consider a position $p \in \mathcal{F}(\text{pref})$. Suppose that we can derive that a match state s with $L(s) = p$, would be f -redundant during the construction. Then p can be removed from `workF`. From Lemma 3.10 it follows that removing redundant positions from `workF` is easy in the absence of consistency partitions. But when we have partial information about term (un)equality recorded in E and N , then it could be that the function symbols of many positions in `workF` are already known.

Example 5.7. Consider the patterns $\ell_1 : f(x, x)$ and $\ell_2 : f(s^{999}(x), s^{999}(y))$ where $s^{999}(x)$ denotes the application of 999 unary successor symbols to the variable x . In the lucky case, we can detect that $f(t_1, t_2)$ satisfies $t_1 = t_2$. The best strategy to also detect a match for ℓ_2 would be to check whether t_1 matches $s^{999}(x)$. This means that t_2 matches $s^{999}(y)$ as well, so almost half the work of checking the function symbols can be skipped.

Suppose we can derive that a consistency state s with $L(s) = \{p, q\}$ will be \checkmark -redundant. Then (p, q) can be removed from `workC` as well.

Example 5.8. Consider the patterns $\ell_1 : f(x, x)$, $\ell_2 : f(g(a), y)$ and $\ell_3 : f(x, g(a))$ and suppose that the selection function prioritises checking all function symbols. Then a consistency check for the term $f(g(a), g(a))$ would be redundant, since we already have full knowledge of all function symbols.

Suppose we can derive that a consistency state s with $L(s) = \{p, q\}$ will be \times -redundant. Then the partition that gave rise to the comparison of positions p and q can be removed

since it is inconsistent. Since this partition could have given rise to pairs in `workC`, and also to positions in `workF`, these sets should be computed again with a reduced set of partitions and patterns.

Example 5.9. Consider again the patterns of Example 5.8. The term $f(g(a), b)$ only matches ℓ_2 . A consistency check to rule out ℓ_1 is redundant after checking all function symbols, because a mismatching function symbols has already been detected.

These ideas could be captured in a procedure that replaces the first three lines in the ANPMA construction. We expect that it is possible to define a procedure that computes minimal sets `workF` and `workC`, along with a smaller pattern set \mathcal{L}'_r from which all derivable inconsistent patterns have been removed. Note that to ensure correctness this procedure also has to adapt the parameters `pref`, E and N internally in a similar way as the algorithm.

Conjecture 5.10. There is an algorithm `REMOVEDUNDANCIES` that takes the parameters \mathcal{L}_r , `pref`, E and N , and yields a (reduced) pattern set $\mathcal{L}'_r \subseteq \mathcal{L}_r$ and two sets `workF` and `workC` such that: replacing lines 2-4 of `CONSTRUCTANPMA` by

$$(\mathcal{L}'_r, \text{workF}, \text{workC}) := \text{REMOVEDUNDANCIES}(\mathcal{L}_r, \text{pref}, E, N),$$

makes `CONSTRUCTANPMA` yield a correct ANPMA without redundant states, for every pattern set \mathcal{L}_r and every selection function `SELECT`.

Note that the comparison in Example 5.7 could also be useful in the case of *only* the pattern $\ell_2 : f(s^{999}(x), s^{999}(y))$. However, the current technique does not allow us to choose arbitrary positions to compare. Therefore, it could also be interesting to either extend the given patterns with new patterns to allow these choices, or to extend the technique to choose arbitrary positions for consistency checks. Similarly, it might be useful to add a comparison state where one of the pairs is not a position, but rather a fixed term without variables. This could be used to exploit constant time comparisons to further improve the efficiency of the matching procedure, which would be especially useful for switch statements over natural numbers.

We now focus on a final example to show that identifying these redundant states can be non-trivial. However, this example shows that interleaving the matching and consistency states can be advantageous in practice. Consider the following patterns: $\ell_1 : f(x, x)$, $\ell_2 : f(x, f(x, y))$, $\ell_3 : f(x, f(y, x))$, $\ell_4 : f(f(x, y), x)$ and $\ell_5 : f(f(y, x), x)$. These patterns can occur as part of an optimisation where applications of an expensive operation f can be avoided. For example, if the f operator implements set union then $f(x, x)$ represents the case where the set union is computed of two equivalent sets. Similarly $f(x, f(x, y))$ represents the case where set union is applied to a set that already contains the set x .

These patterns are non-linear and therefore we first apply the `rename` function to obtain the corresponding linear pattern and consistency partitions. This result in the following renamed pattern set.

$$\begin{aligned} \ell_1 &: (f(\omega_1, \omega_2), \{\{1, 2\}\}) \\ \ell_2 &: (f(\omega_1, f(\omega_{2.1}, \omega_{2.2})), \{\{1, 2.1\}\}) & \ell_3 &: (f(\omega_1, f(\omega_{2.1}, \omega_{2.2})), \{\{1, 2.2\}\}) \\ \ell_4 &: (f(f(\omega_{1.1}, \omega_{1.2}), \omega_2), \{\{1.1, 2\}\}) & \ell_5 &: (f(f(\omega_{1.1}, \omega_{1.2}), \omega_2), \{\{1.2, 2\}\}) \end{aligned}$$

If we consider the APMA for the set of linear patterns: $f(\omega_1, \omega_2)$, $f(\omega_1, f(\omega_{2.1}, \omega_{2.2}))$ and $f(f(\omega_{1.1}, \omega_{1.2}), \omega_2)$ then we can easily see that it takes three steps to reach any of the final states. Namely, we check the positions ϵ , one and two for the occurrence of the head symbol f and then we obtain the subset of patterns that match, which can be either $\{\ell_1\}$, $\{\ell_1, \ell_2, \ell_3\}$, $\{\ell_1, \ell_4, \ell_5\}$ or $\{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5\}$. After that a consistency automaton or naive consistency

check requires exactly one comparison for each element in the set to determine the set of consistent partitions.

However, the consistency automaton contains some redundancies. For example, whenever the partition of ℓ_1 is consistent w.r.t. a term t it cannot be the case that any of the other partitions are consistent w.r.t. term t , because a term cannot be equal to any of its subterms. Formally, this means that $t \neq t[i]$ for any natural number i , which can be proven by structural induction on terms. We can generalise this statement such that for any positions p and $r \neq \epsilon$ it holds that $t[p.r] \neq t[p]$.

This observation can be used to avoid the previously mentioned redundancy. If for any term t it holds that $t[1] = t[2]$ then only partition $\{1, 2\}$ can be consistent with respect to t . Furthermore, if we know that for a term $t[1.1] = t[2]$ then $t[1.1.1] = t[2.1]$ by definition of equality and since $t[1] \neq t[1.1.1]$ by the observation above it follows that $t[1] \neq t[2.1]$. Therefore, we can conclude that if $t[1.1] = t[2]$ then only patterns ℓ_4 and ℓ_5 can match. Similarly, if $t[1] = t[2.1]$ then only patterns ℓ_2 and ℓ_3 can match. Using these observations we can construct the ANPMA without redundancies that is shown in Figure 6.

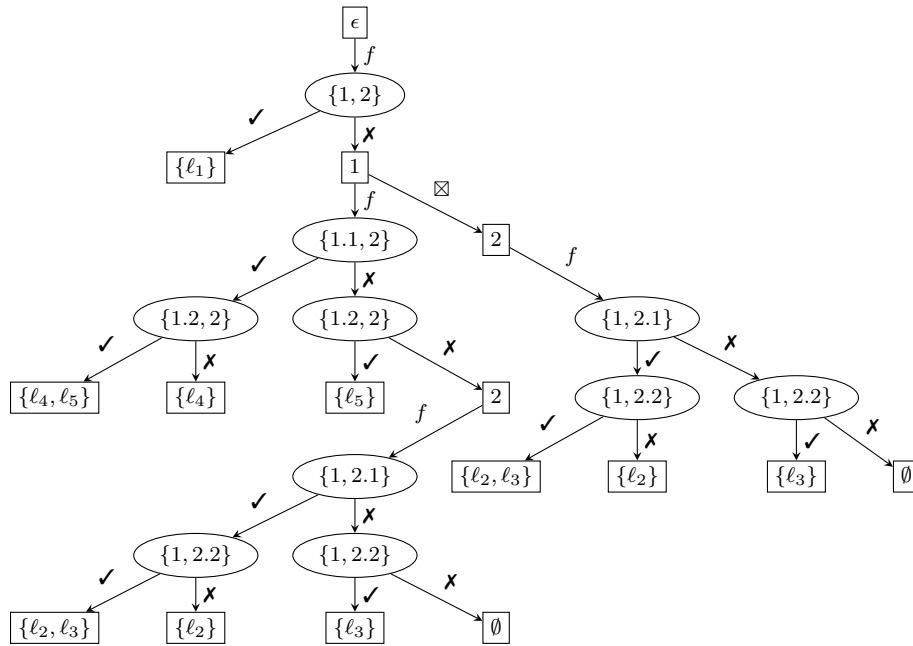


FIGURE 6. The pruned ANPMA for the patterns $\ell_1 : f(x, x)$, $\ell_2 : f(x, f(x, y))$, $\ell_3 : f(x, f(y, x))$, $\ell_4 : f(f(x, y), x)$ and $\ell_5 : f(f(y, x), x)$.

This ANPMA is both smaller in the number of states when compared to an APMA followed by individual CAs. Furthermore, its evaluation depth for all terms that match the patterns ℓ_1, ℓ_2 and ℓ_3 is strictly smaller than without interleaving. For the patterns ℓ_4 and ℓ_5 the evaluation depth remains the same. Therefore, this example shows that interleaving the two phases and removing redundancies can yield a more efficient non-linear matching procedure in practice.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a formal proof for the correctness of APMA's. Furthermore, we introduced as a deterministic automaton to perform the consistency checking, from which some redundant states could be removed by taking the previous choices into account. These two automata are then combined to obtain an ANPMA which could be evaluated by only performing comparisons and taking the corresponding outgoing edge.

ANPMA's offer a formal platform to study the relations between linear pattern matching and consistency checking. There are still some questions that have arisen from this work. As mentioned in the previous section, the current ANPMA construction algorithm can contain redundant states. We expect that there is an optimisation function as described in Conjecture 5.10 that takes care of this problem. But it might just as well be the case that this problem is undecidable altogether.

Secondly we did not study selection functions in this work. All three automaton construction algorithms in this paper are parametrised in a selection function that decides for each node what will happen next. We have shown that all constructions yield correct automata for every selection function, with the side note that the selection indeed yields an element from its input set. The size of all three kinds of automata depends heavily on the selection function that is used. For APMA's some selection functions have already been studied in [SRR95].

Finally, it would be interesting to implement this approach. This work is a theoretical approach to ultimately reduce the number of steps that matching requires in for example a term rewrite engine. However, many of the existing pattern matching problems do not support non-linear pattern matching as discussed in the introduction. It would be interesting to find out whether exploiting $\mathcal{O}(1)$ term equality checking is worth the construction time and in particular the identification of redundant states in practice.

ACKNOWLEDGEMENTS

We would like to thank Jan Friso Groote, Bas Luttik and Tim Willems for their feedback and discussion. This work was supported by the TOP Grants research programme with project number 612.001.751 (AVVA), which is (partly) financed by the Dutch Research Council (NWO).

REFERENCES

- [Car84] L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217. ACM, 1984.
- [Chr93] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993.
- [CS76] Douglas Comer and Ravi Sethi. Complexity of trie index construction (extended abstract). In *FOCS*, pages 197–207. IEEE Computer Society, 1976.
- [FM01] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP*, pages 26–37. ACM, 2001.
- [Grä91] Albert Gräf. Left-to-right tree pattern matching. In *RTA*, volume 488 of *LNCS*, pages 323–334. Springer, 1991.
- [Gra95] P. Graf. Substitution tree indexing. In J. Hsiang, editor, *Rewriting Techniques and Applications*, pages 117–131, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

- [HB20] Gabriel Hondet and Frédéric Blanqui. The new rewriting engine of dedukti (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPICs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In Eijiro Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 35–46. ACM, 2008.
- [McC92] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, Oct 1992.
- [SRR95] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal of Computing*, 24(6):1207–1234, 1995.
- [SRV01] R. Sekar, I.V. Ramakrishnan, and A. Voronkov. Chapter 26 - term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 1853 – 1964. North-Holland, Amsterdam, 2001.
- [Vor95] A. Voronkov. The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.
- [vW07] M. van Weerdenburg. An account of implementing applicative term rewriting. *Electronic Notes in Theoretical Computer Science*, 174(10):139–155, 2007.

APPENDIX A. PROOF DETAILS OF SECTION 3

A.1. Proof details of Lemma 3.5.

Proof. By induction on the length of $\text{path}(s)$. If there are no pairs in $\text{path}(s)$ then it must be that $s = s_0$. From $\text{pref}(s_0) = \omega_\epsilon$ it follows that $\mathcal{L}(s_0) = \mathcal{L}$. Then the base case follows from $\mathcal{L}_t \subseteq \mathcal{L} = \mathcal{L}(s_0) = \mathcal{L}(s)$.

Let s be an arbitrary state and suppose that $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s)$ and assume the induction hypothesis $\mathcal{L}_t \subseteq \mathcal{L}(s)$. Now suppose $\text{MATCH}(M, t, s) = \text{MATCH}(M, t, s')$ where $s' = \delta(s, f)$ for some $f \in \mathbb{F}_\boxtimes$ and let $L(s) = p$.

- If $f \in \mathbb{F}$ then $\text{pref}(s') = \text{pref}(s)[p/f(\omega_{p.1}, \dots, \omega_{p.\text{ar}(f)})]$. By definition of MATCH we know that $\text{head}(t[p]) = f$.

Let $\ell \in \mathcal{L}_t$. We show that ℓ unifies with $\text{pref}(s')$. We know that $\ell \leq t$ by assumption. From the induction hypothesis it follows that ℓ unifies with $\text{pref}(s)$. So there is a term u such that $\ell \leq u$ and $\text{pref}(s) \leq u$. Then we distinguish two cases.

- If $\ell[p']$ is a variable for some $p' \sqsubseteq p$ then ℓ unifies with $\text{pref}(s')$.
- If $\text{head}(\ell[p])$ is a function symbol then by $\ell \leq t$ it must be that $\text{head}(\ell[p]) = f$, so ℓ unifies with $\text{pref}(s')$.
- If $f = \boxtimes$ then $\text{pref}(s') = \text{pref}(s)[p/\boxtimes]$. By definition of MATCH we know that $\delta(s, \text{head}(t[p]))$ is undefined.

From the construction algorithm we then know that there is no pattern $\ell \in \mathcal{L}(s)$ such that $\text{head}(\ell[p]) \in \mathbb{F}$ and there is at least one pattern $\ell \in \mathcal{L}(s)$ such that $\ell[p']$ is a variable for some position $p' \sqsubseteq p$.

Let $\ell \in \mathcal{L}_t$. By induction hypothesis we know that ℓ unifies with $\text{pref}(s)$. We show that ℓ unifies with $\text{pref}(s')$ by showing that $\ell[p'] = \omega_{p'}$ for some position $p' \sqsubseteq p$.

- Suppose that $\ell[p]$ exists. Since $\ell \leq t$ and $\text{head}(t[p]) \neq \text{head}(\ell[p])$ it must be that $\ell[p] = \omega_p$.
- Suppose that $\ell[p]$ does not exist. Pick the lowest position p' such that $p' \sqsubset p$ and $\ell[p']$ exists and assume for a contradiction that $\text{head}(\ell[p']) = f$ for some function symbol f . Then it must be that $\text{head}(\text{pref}(s)[p']) = f$ by the induction hypothesis. However,

$\text{pref}(s)[p]$ exists and from $p' \sqsubset p$ it follows that $\ell[p]$ has subterms of the function symbol f , which contradicts the assumption that p' is the lowest position strictly higher than p . So $\ell[p'] = \omega_{p'}$. \square

A.2. Proof details of Lemma 3.6.

Proof. a) We show that $\text{MATCH}(M, t, s_0) \neq L(s)$ for all final states s . Let s_f be an arbitrary final state and pick some pattern $\ell \in L(s_f)$. By assumption $\ell \not\leq t$ and by Proposition 3.1 it follows that there is a position p and a function symbol $f \in \mathbb{F}$ such that $\text{head}(\ell[p]) = f$ and $\text{head}(t[p]) \neq f$. By Lemma 3.4 it must be that $\text{head}(\text{pref}(s)[p]) = f$, by which there must be a pair $(s_i, f) \in \text{path}(s)$. Since MATCH is a function we have $\text{MATCH}(M, t, s_0) = \text{MATCH}(M, t, s_i) = \text{MATCH}(M, t, s_f)$. Then, by definition of MATCH we obtain $\text{head}(t[p]) = f$, a contradiction.

b) Let $\ell \in \mathcal{L}_t$. We prove that for all s such that $\text{MATCH}(M, s_0, t) = \text{MATCH}(M, s, t)$, we have that $\delta(s, \text{head}(t[L(s)]))$ or $\delta(s, \boxtimes)$ is defined.

Suppose that $\text{MATCH}(M, s_0, t) = \text{MATCH}(M, s, t)$. From Lemma 3.5 it follows that $\ell \in \mathcal{L}(s)$. If $\text{head}(\ell[L(s)]) = f$ for some function symbol f then the construction algorithm created an f -transition to a new state, by which $\delta(s, f)$ exists. Otherwise if $\text{head}(\ell[L(s)])$ does not exist then by $\ell \leq t$ there must be a position $p \sqsubset L(s)$ such that $\ell[p] = \omega_p$. In that case a \boxtimes -transition is created and hence $\delta(s, \boxtimes)$ exists.

By definition of MATCH we then have that $\text{MATCH}(M, s_0, t)$ cannot yield the empty set, so it must terminate in a final state. \square

A.3. Proof details of Lemma 3.7.

Proof. Since $L(s_f) = \mathcal{L}(s_f)$ we know that $\mathcal{L}_t \subseteq L(s_f)$ by Lemma 3.5. It only remains show that $L(s_f) \subseteq \mathcal{L}_t$. Since s_f is a final state we have that $L(s_f) = \{\ell \in \mathcal{L} \mid \ell \leq \text{pref}(s_f)\}$. Suppose for a contradiction that there is some $\ell \leq \text{pref}(s_f)$ such that $\ell \not\leq t$. Then there is a position p such that $\text{head}(\ell[p]) \in \mathbb{F}$ and $\text{head}(t[p]) \neq \text{head}(\ell[p])$. We have $\text{head}(\ell[p]) = \text{head}(\text{pref}(s_f)[p])$ by assumption. So, there is a pair (s_i, f_i) in $\text{path}(s_f)$ such that $L(s_i) = p$. By definition of MATCH we then have $\text{head}(t[p]) = f_i = \text{head}(\ell[p])$, a contradiction. \square