Adaptive Non-Linear Pattern Matching Automata

Rick Erkens
Eindhoven University of Technology, The Netherlands
r.j.a.erkens@tue.nl

Maurice Laveaux
Eindhoven University of Technology, The Netherlands
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 that its evaluation depth can be shorter than two-phase approaches.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern matching, Term indexing, Tree automata

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.20

Acknowledgements We would like to thank Jan Friso Groote, Bas Luttik and Tim Willemse for their feedback and discussion. We would also like to thank the anonymous reviewers for their effort and constructive remarks and questions.

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 reduction steps invalidate other matches. A root pattern matching algorithm can 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 [7] have been proposed to determine matching patterns efficiently. Adaptive pattern matching automata [8] (APMA) are tree-like data structures that are constructed from a set of patterns. By using such 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 not restricted to a fixed traversal such as a left-to-right traversal in [4]. The size of an APMA is worst-case exponential in the size of the pattern set, but in practice its size is typically smaller and this preprocessing 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 [4, 8] 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.* Performing matching and consistency checking separately does not yield the optimal matching time. Therefore we extend the existing APMA with consistency checking on the fly. Our extension preserves the adaptive traversal of [8] and allows information about the matching step to influence the consistency checking, and the other way around.

We introduce *consistency automata* (CA) to perform the variable consistency check efficiently for a set of patterns. The practical use of this automaton is based on similar observations as the pattern matching automata: 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. Then we introduce *adaptive non-linear pattern matching automata* (ANPMA), 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 ANPMA yield a correct matching algorithm for non-linear patterns. To this end we also give a correctness proof for the APMA approach from [8], which was not given in the original work.

We compare this work with other term indexing techniques. Most techniques use tree-like data structures with deterministic [1, 4, 8, 9] or non-deterministic [3, 2, 6, 10, 5] 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.

Not all techniques support matching non-linear patterns. *Discrimination trees* [6], *substitution trees* [5] and *match trees* [9] can be extended with on-the-fly consistency checks for matching non-linear patterns. Their evaluation strategy however is restricted to preorder evaluation and variable consistency must be checked whenever a variable which has already been bound occurs at the current evaluated position of the pattern. We have also considered *code trees* [10], which also have preorder evaluation with backtracking. These allow consistency checks to occur at different places. All three approaches might inspect the same position multiple times due to backtracking. The ANPMAs introduced in this paper mitigate these issues: 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 no redundant checks.

2 Preliminaries

In this section the preliminaries of first-order terms and the 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 $A \to B$, $A \rightharpoonup B$ and $A \hookrightarrow B$ 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 $\bot$ for elements in its domain for which it is undefined. Furthermore, we assume the existence of an index set $\mathcal{I}$ and use $A \times \mathcal{I}$ to denote the indexed family with elements denoted by $i : a$ for $a \in A$ and $i \in \mathcal{I}$.

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 $\mathsf{ar}(f)$, 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 $\mathbb{T}_\Sigma$, is defined as the smallest set such that $\mathbb{V} \subseteq \mathbb{T}_\Sigma$ and whenever $t_1, \ldots, t_n \in \mathbb{T}_\Sigma$ and $f \in \mathbb{F}_n$, then also $f(t_1, \ldots, t_n) \in \mathbb{T}_\Sigma$. 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 $\mathsf{head}(x) = x$ for a variable x and $\mathsf{head}(f(t_1, \ldots, t_n)) = f$ for a term $f(t_1, \ldots, t_n)$. We use $\mathsf{vars}(t)$ to denote the set of variables that occur in term t. A term for which $\mathsf{vars}(t) = \emptyset$ is called a *ground term.* A *pattern* is a term of the form $f(t_1, \ldots, t_n)$. 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 $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ 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 $f(t_1, \ldots, t_n) =_\omega f(t'_1, \ldots, t'_n)$ 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 $f(t_1, \ldots, t_n)^\sigma = f(t_1^\sigma, \ldots, t_n^\sigma)$. 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 $t[p] = f(t_1, \ldots, t_n)$ 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 $f(x, y)[1.1]$. 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 $f(t_1, \ldots, t_n)[(i.p)/u] = f(t_1, \ldots, t_i[p/u], \ldots, t_n)$. 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 term $f(\omega_{1.1})$ is 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 1.** *Let $\mathcal{L} \subseteq \mathbb{T}_\Sigma$ be a set of patterns. A function $match_\mathcal{L} : \mathbb{T}_\Sigma \to 2^{\mathbb{T}_\Sigma}$ 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.

▶ **Proposition 2.** *Let ℓ and ℓ' be linear patterns. We have that $\ell \leq \ell'$ if and only if for all positions p: if $\mathit{head}(\ell[p]) \in \mathbb{F}$ then $\mathit{head}(\ell[p]) = \mathit{head}(\ell'[p])$.*

A naive matching algorithm for linear patterns follows directly from this proposition: to find all matches for term t one can check the proposition for every pattern separately. However, for a *set* of 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. This is the purpose of so-called *term indexing techniques* [7]. Sekar et al. [8] 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 advantage of using an APMA over the naive approach is that for every input term, every position is inspected at most once.

We present the evaluation and construction procedures of APMAs slightly differently compared to the presentation by Sekar et al. APMAs are state machines in which every state is a matching state, which is labelled with a position, or final state, which is labelled with a set of patterns. Matching 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.** *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;*
- *$\delta : S_M \times \mathbb{F}_\boxtimes \rightharpoonup S$ is a partial transition function;*
- *$L = L_M \uplus L_F$ is a state labelling function with $L_M : S_M \to \mathbb{P}$ and $L_F : S_F \to 2^{\mathbb{T}_\Sigma}$*
- *$s_0 \in S_M$ 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 $\delta(s, f) = s_0$.

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 1 shows an APMA that can be used to decide which of these patterns match. In addition to the position label on every matching state, it also displays the term that represents what has been matched so far. That is, in the state labelled with position 2, only the function symbol f has been inspected. The term $f(\omega_1, \omega_2, \omega_3)$ represents that f has been inspected and the variables at positions 1, 2 and 3 represent that these positions have not been inspected. We refer to this term as a *prefix*. Prefixes are not a part of the APMA; they are included for comprehensiveness only. Later they will aid in the construction algorithm and the correctness proof.

The function Match below defines the evaluation of an APMA on a term. Upon reaching a final state $s \in S_F$ the evaluation yields the set of terms $L_F(s)$. In a matching state $s \in S_M$ the head symbol $\mathsf{head}(t[L_M(s)])$ is examined. If there is an outgoing transition labelled with this head symbol then evaluation continues in the resulting state; otherwise the $\boxtimes$-transition is taken. Whenever there is no outgoing $\boxtimes$-transition then there is no match and the evaluation returns the empty set as a result.

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

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

If we consider the APMA M of Figure 1 and let initial state s_0 be the topmost state in the figure. We have $\textsc{Match}(M, f(a, b, a), s_0) = \{f(a, b, x)\}$ and $\textsc{Match}(M, f(b, b, b), s_0) = \emptyset$. The term $f(c, b, b)$ will yield the pattern set $\{f(c, b, x)\}$.

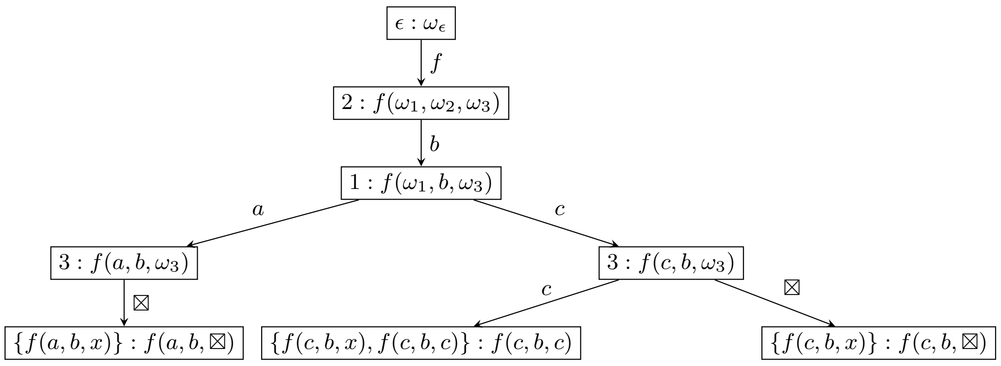

$\epsilon : \omega_\epsilon$
f
$2 : f(\omega_1, \omega_2, \omega_3)$
b
$1 : f(\omega_1, b, \omega_3)$
a
c
$3 : f(a, b, \omega_3)$
$3 : f(c, b, \omega_3)$
$\boxtimes$
c
$\boxtimes$
$\{f(a, b, x)\} : f(a, b, \boxtimes)$
$\{f(c, b, x), f(c, b, c)\} : f(c, b, c)$
$\{f(c, b, x)\} : f(c, b, \boxtimes)$

Figure 1 An APMA constructed from the patterns given above.

In Algorithm 1 the APMA construction is defined. Intuitively CONSTRUCT creates the APMA from root to leaf based on the pattern set $\mathcal{L}$ and the selection function SELECT. For convenience we also assume that all patterns in $\mathcal{L}$ are position-annotated. In later sections we drop this assumption in order to treat 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 prefix associated with each state plays an important role during construction. 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. 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 the 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 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 $\mathcal{F}(\mathsf{pref})$. 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.

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. We make use of the following auxilliary definitions. A *path* to s_n is a sequence of state and function symbol pairs $(s_0, f_0), \ldots, (s_{n-1}, f_{n-1}) \in S_M \times \mathbb{F}_\boxtimes$ 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 $\mathsf{path}(s)$. A matching state s is *top-down* iff $L(s) = \epsilon$ or there is a pair (s_i, f_i) in $\mathsf{path}(s)$ with $L(s_i).j = L(s)$ for some $1 \leq j \leq \mathsf{ar}(f_i)$. State s is *canonical* iff there are no two states in $\mathsf{path}(s)$ that are labelled with the same position. Finally we say that an APMA is *well-formed* iff all matching 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 $\mathsf{pref}(s)$. It is constructed inductively for well-formed APMAs by $\mathsf{pref}(s_0) =$

Algorithm 1 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 $\mathsf{pref} = \omega_\epsilon$.

```
procedure CONSTRUCT($\mathcal{L}$, SELECT, $M$, $s$, pref)
    $\mathcal{L}' := \{\ell \in \mathcal{L} \mid \ell$ unifies with pref$\}$
    work := $\mathcal{F}$(pref)
    if work = $\emptyset$ then
        $M := M[S_F := (S_F \cup \{s\}), L_F := L_F[s \mapsto \mathcal{L}']]$
    else
        pos := SELECT(work)
        $M := M[S_M := (S_M \cup \{s\}), L_M := L_M[s \mapsto \mathsf{pos}]]$
        $F := \{f \in \mathbb{F} \mid \exists \ell \in \mathcal{L}' : \mathsf{head}(\ell[\mathsf{pos}]) = f\}$
        for $f \in F$ do
            $M := M[\delta := \delta[(s, f) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
            $M :=$ CONSTRUCT($\mathcal{L}$, SELECT, $M$, $s'$, $\mathsf{pref}[\mathsf{pos} := f(\omega_{\mathsf{pos}.1}, \dots, \omega_{\mathsf{pos}.\mathsf{ar}(f)})]$)
        if $\exists \ell \in \mathcal{L}' : \exists \mathsf{pos}' \sqsubseteq \mathsf{pos} : \mathsf{head}(\ell[\mathsf{pos}']) \in \mathbb{V}$ then
            $M := M[\delta := \delta[(s, \boxtimes) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
            $M :=$ CONSTRUCT($\mathcal{L}$, SELECT, $M$, $s'$, $\mathsf{pref}[\mathsf{pos} := \boxtimes]$)
    return $M$
```

ω_ϵ and if $\delta(s_i, f) = s_{i+1}$ then $\mathsf{pref}(s_{i+1}) = \mathsf{pref}(s_i)[L(s_i)/f(\omega_{L(s_i).1}, \dots, \omega_{L(s_i).\mathsf{ar}(f)})]$. Similarly, we denote the patterns of state s for all states by $\mathcal{L}(s) = \{\ell \in \mathcal{L} \mid \ell \text{ unifies with } \mathsf{pref}(s)\}$. Lastly we use an arbitrary function SELECT : $2^{\mathbb{P}} \to \mathbb{P}$ such that for all sets of positions work we have SELECT(work) $\in$ work.

▶ **Lemma 4.** *For all finite, non-empty sets of patterns $\mathcal{L}$ we have that the procedure* CONSTRUCT$(\mathcal{L}, \text{SELECT}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon)$ *terminates and yields a well-formed APMA* $M = (S, \delta, L, s_0)$.

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 CONSTRUCT$(\mathcal{L}, \text{SELECT}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon)$. Furthermore, let t be an arbitrary term and let $\mathcal{L}_t = \{\ell \in \mathcal{L} \mid \ell \leq t\}$.

The following lemmas state some claims and invariants about CONSTRUCT and its relation to MATCH. The proofs are rather tedious and are attached in the appendix.

▶ **Lemma 5.** *For every every final state s: (a) the set $L(s)$ is non-empty, (b) $\mathsf{pref}(s)$ is a ground term, and (c) for all $\ell \in L(s)$ we have $\ell \leq \mathsf{pref}(s)$. Moreover (d) for every pattern $\ell \in \mathcal{L}$ there is at least one final state s with $\ell \in L(s)$.*

▶ **Lemma 6.** *For all states s such that* MATCH$(M, t, s_0) =$ MATCH(M, t, s) *it holds that* $\mathcal{L}_t \subseteq \mathcal{L}(s)$.

▶ **Lemma 7.** *It holds that:*

a) *If $\mathcal{L}_t = \emptyset$ then* MATCH$(M, t, s_0) = \emptyset$*;*

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

▶ **Lemma 8.** *If* MATCH$(M, t, s_0) =$ MATCH(M, t, s_f) *for some final state s_f then* $L(s_f) = \mathcal{L}_t$.

▶ **Theorem 9.** *Then $\lambda t.$*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 7 we get that MATCH$(M, t, s_0) = \emptyset = \mathcal{L}_t$ as required. If $\mathcal{L}_t$ is non-empty then by Lemma 7 we have that MATCH$(M, t, s_0) =$ MATCH(M, t, s_f) for some final state s_f. Then by definition of MATCH we get MATCH$(M, t, s_f) = L(s_f)$. From Lemma 8 it follows that $L(s_f) = \mathcal{L}_t$, by which we can conclude MATCH$(M, t, s_0) = \mathcal{L}_t$. Hence $\lambda t.$MATCH(M, t, s_0) is a linear matching function for $\mathcal{L}$. ◀

4 Consistency Checking

As mentioned in other literature [4, 7, 8] a linear matching algorithm can be used to solve the non-linear matching problem by transforming the patterns and checking so-called *variable consistency* after the matching phase. This is required because a variable which occurs at multiple positions can only be assigned a single value in the matching substitution. First, a transformation step ensures that all input terms are changed into linear patterns by *renaming* different occurrences of the variables in the non-linear patterns. The linear matching algorithm can then be used to solve part of the non-linear matching problem. Finally, a consistency check is performed to remove the linear patterns for which the substitution that witnesses the match is not valid for the original patterns. We first focus on efficiently deciding this variable consistency step.

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, 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 that are forgotten when a non-linear pattern is renamed. For this purpose we introduce *consistency classes* [7]. This is a set of positions with the following notion.

▶ **Definition 10.** *Given a term t and a consistency class $C \subseteq \mathbb{P}$ we say that t is* consistent *w.r.t. 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 classes $\{1, 2\}$, $\{3, 4, 5\}$ and $\{6\}$. This means that for the input term $t = f(t_1, \ldots, t_6)$ that both $t[1] = t[2]$ 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 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 w.r.t. a consistency partition P iff t is consistent w.r.t. C for every $C \in P$.

First, we illustrate the renaming procedure by means of an example. For the purpose of renaming, partitions of the fringe of a pattern are sufficient. 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 $\mathsf{id}[\omega_1 \mapsto a, \omega_2 \mapsto a, \omega_3 \mapsto b]$, but $f(a, a, b)$ is only consistent w.r.t. partition P_1. Therefore, 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 11.** *The term rename function $\mathsf{rename} : \mathbb{T}_\Sigma \to (\mathbb{T}_{\Sigma_\mathbb{P}} \times 2^{2^{\mathbb{P}}})$ is defined as*

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

where $\mathsf{rename}_1(t, \epsilon) : (\mathbb{T}_\Sigma \times \mathbb{P}) \to \mathbb{T}_{\Sigma_\mathbb{P}}$ *renames the variables of the given term to position variables, which is defined below.*

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

Note that for linear patterns the result is a position annotated term with trivial consistency classes. We show a number of characteristic properties of the rename function which are essential for the non-linear matching algorithm.

▶ **Lemma 12.** *For all terms* $t \in \mathbb{T}_\Sigma$ *if* $(t', P) = \mathsf{rename}(t)$ *then:*

- $t =_\omega t'$*;*
- *for all* $p \in \mathcal{F}(t)$*:* $t'[p] = \omega_p$*;*
- *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*.*

For the variable consistency phase a straightforward implementation follows directly from Definition 10. Let $P = \{C_1, \ldots, C_n\}$ be a 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 IS-CONSISTENT(t, P) to denote this naive algorithm. For a set of partitions $\{P_1, \ldots, P_m\}$ the (naive) consistency check requires exactly $\sum_{1 \leq j \leq m} \sum_{C \in P_j} |C|$ comparisons if t is consistent w.r.t. P.

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, indexed by elements from $\mathcal{I}$, and adapting the rename function to preserve the corresponding indices. Now, when given an indexed linear pattern that resulted from renaming we can identify the corresponding original pattern by its index. The following lemma follows directly from the third property of Lemma 12.

▶ **Lemma 13.** *Let* $\mathcal{L} \subseteq \mathbb{T}_\Sigma \times \mathcal{I}$ *be a set of patterns and let* $\mathcal{L}_r \subseteq \mathbb{T}_{\Sigma_\mathbb{P}} \times 2^{2^\mathbb{P}} \times \mathcal{I}$ *be the set of linear patterns and corresponding consistency partitions resulting from renaming; i.e.,* $\mathcal{L}_r = \{\mathsf{rename}(l) \mid l \in \mathcal{L}\}$*. Let* $\mathsf{match\text{-}linear} : \mathbb{T}_\Sigma \times 2^{\mathbb{T}_\Sigma} \times \mathcal{I} \to 2^{\mathbb{T}_\Sigma} \times \mathcal{I}$ *be a linear matching function that preserves indices. For any term* $t \in \mathcal{L}$ *we define* $\mathsf{match} : (\mathbb{T}_\Sigma \times (\mathbb{T}_{\Sigma_\mathbb{P}} \times 2^{2^\mathbb{P}} \times \mathcal{I})) \to \mathbb{T}_\Sigma$ *as:*

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

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

4.2 Consistency Automata

In this section, we are only going to focus on solving the consistency checking efficiently and later on we show that the matching time can be further improved by interleaving the choices. Consider the consistency partitions $P_1 = \{\{1, 2\}, \{3\}\}$, $P_2 = \{\{1, 3\}, \{2\}\}$ and $P_3 = \{\{1, 2, 3\}\}$ again. We would expect that similarly to an APMA we can use the fact that comparisons of overlapping partitions can be used to determine the subset of all consistent partitions directly. This means 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. For this reason,

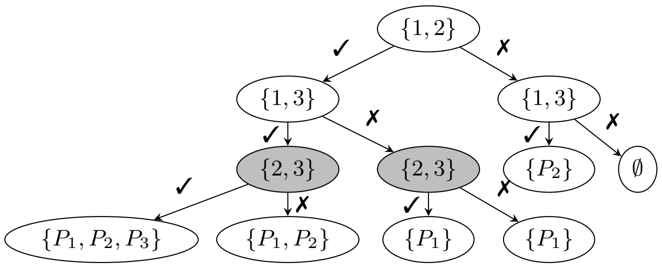


Figure 2 The CA for the partitions $P_1 = \{\{1,3\},\{2\}\}$, $P_2 = \{\{1,2\},\{3\}\}$ and $P_3 = \{\{1,2,3\}\}$ where positions 1 and 2 are compared first, followed by 1 and 3 and finally 2 and 3. The grey states are redundant and can be removed as shown in later steps.

we define *consistency automata* which are constructed from a set of consistency partitions. Each state of this automaton is labelled with a pair of positions that should be compared. Similar labelling is also present in other matching algorithms [10], but not presented as a separate automaton. Afterwards, we show that redundant comparisons can be removed such that this example requires at most two comparisons.

A consistency automaton, abbreviated CA, is a state machine where every state is a consistency state, which is labelled with a pair of positions, or a final state, which is labelled with set of partitions. The transitions are labelled with either ✓ or ✗ 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.

▶ **Definition 14.** *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*;*
- $\delta : (S_C \times \{✓, ✗\}) \to S$ *is a transition function;*
- $L = L_C \uplus L_F$ *is a state labelling function with* $L_C : S_C \to \mathbb{P}^2$ *and* $L_F : S_F \to 2^{\mathcal{I}}$*;*
- $s_0 \in S$ *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 2 shows a CA that can be used to decide the consistency of a given term t w.r.t. any of these partitions. If the consider the state labelled with $\{1,2\}$ the subterms $t[1]$ and $t[2]$ are compared. Whenever these are equal the evaluation continues with the ✓-branch and it continues with the ✗-branch otherwise. If a final state (labelled with partitions) is reached then t is consistent w.r.t. these partitions by construction.

The evaluation function of a CA for the input term and a given state, starting with the initial state, is defined below. First, it checks whether the current state is final, in which case 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 ✓-transition; otherwise it proceeds with the state of the outgoing ✗-transition.

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

The construction procedure of a CA is defined in Algorithm 2. 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 to compare.

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 $\subseteq\in$ for the composition of $\subseteq$ and $\in$; 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 ✓-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 ✗-transition is created.

Algorithm 2 Given a set of partitions $P = \{P_1,\dots,P_n\}$ then CONSTRUCT-CA$(P,\textsc{Select})$ computes a CA using CONSTRUCT-CA$(P,\textsc{Select},(\emptyset,\emptyset,\emptyset,s_0),s_0,\emptyset,\emptyset)$ that can be used to evaluate the consistent partitions using EVAL-CA.

```
1:  procedure CONSTRUCT-CA($P, \textsc{Select}, M, s, E, N$)
2:      $P' := \{P_i \in P \mid \neg\exists C \in P_i : \exists\{p,q\} \in N : p,q \in C\}$
3:      work $:= \{\{p,q\} \in \mathbb{P}^2 \mid \{p,q\} \subseteq\in P_i \wedge P_i \in P'\} \setminus E$
4:      if work $= \emptyset$ then
5:          $M := M[S_F := (S_F \cup \{s\}), L_F := L_F[s \mapsto P']]$
6:      else
7:          $\{p,q\} := \textsc{Select}(\text{work})$
8:          $M := M[S_C := (S_C \cup \{s\}), L_C := L_C[s \mapsto \{p,q\}]]$
9:          $M := \textsc{construct-ca}(P, \textsc{Select}, M[\delta := \delta[(s,\checkmark) \mapsto s']], s', E \cup \{\{p,q\}\}, N)$     where $s'$ is a fresh unbranded state w.r.t. $M$.
10:         $M := \textsc{construct-ca}(P, \textsc{Select}, M[\delta := \delta[(s,\text{✗}) \mapsto s']], s', E, N \cup \{\{p,q\}\})$     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 how some 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 17. 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 SELECT : $2^{\mathbb{P}^2} \to \mathbb{P}^2$ be any selection function such that SELECT(work) $\in$ work for all non-empty 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. Again, the complete proofs are present in the appendix.

▶ **Lemma 15.** *The procedure* CONSTRUCT-CA$(P, \textsc{Select})$ *terminates.*

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

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 EVAL-CA(M, t, s_0) = EVAL-CA(M, t, s_n) it holds that that $(E(s_n), N(s_n)) \models t$.

▶ **Lemma 16.** *Let $M = (S, \delta, L, s_0)$ be the result of* CONSTRUCT-CA$(P, \textsc{Select})$*. Then M is well-formed.*

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 17.** *Let $M = (S, \delta, L, s_0)$ be the result of* CONSTRUCT-CA$(P, \textsc{Select})$ *then*

- *for all terms $t \in \mathbb{T}_\Sigma$ we have $P' =$ EVAL-CA(M, s_0, t) for some $P' \subseteq 2^{2^{\mathbb{P}}}$, and*
- *for all $P_j \in P$ it holds that $P_j \in P'$ iff the term t is consistent w.r.t. P_j.*

Proof. We have already shown termination of the construction procedure in Lemma 15. Let P' be the set of partitions returned by EVAL-CA(M, t, s_0), let $P_i \in P$ be any partition and EVAL-CA(M, t, s_0) = EVAL-CA(M, t, s_n) for some final state $s_n \in S_F$. By Lemma 16 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 $t[p] \neq t[q]$.

$\Longrightarrow$) Assume that $P_j \in P'$. For all p, q such that $\{p, q\} \subseteq\in P_j$ it holds that $\{p, q\} \in E(s_n)$ as $\mathsf{work}(s_n)$ 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.

$\Longleftarrow$) 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 $\{p, q\} \subseteq\in P_j$ 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. ◀

4.4 Efficiency

Given a CA M and a term t we define the *evaluation depth*, denoted by $\mathrm{ED}(M, t)$, as the number of recursive EVAL-CA calls made to reach the final state. The size, denoted by $|M|$, is given by the number of states of M. The number of transitions is omitted as each non-final state has exactly two outgoing transitions. We define a notion of relative efficiency that compares the evaluation depth of two automata for all input terms.

▶ **Definition 18.** *Given two consistency automata $M = (S, \delta, L, s_0)$ and $M' = (S', \delta', L', s'_0)$ for a set of consistency partition P. We say that $M \preceq M'$ iff for all terms $t \in \mathbb{T}_\Sigma$ it holds that $ED(M, t) \leq ED(M', t)$.*

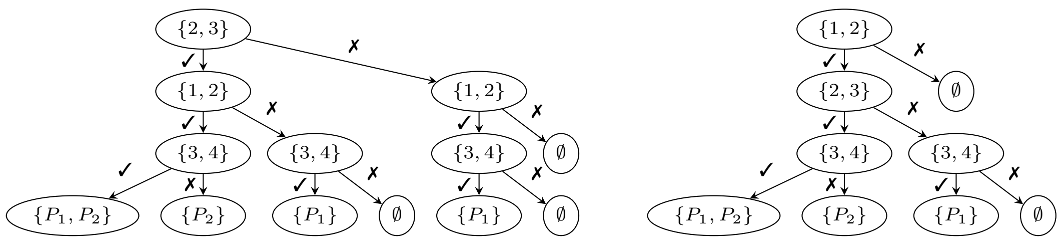


Figure 3 Two CA for the partitions $P_1 = \{\{1,2\},\{3,4\}\}$ and $P_2 = \{\{1,2,3\}\}$. The CA on the left chooses $\{2,3\}$ first. However, as shown on the right selecting $\{1,2\}$ first removes both partitions, and leads to a smaller CA.

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 relative efficiency and size of the resulting CA as shown in Figure 3.

Changing the selection function does not necessarily result in the most efficient (equivalent) CA. If we consider Figure 2 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 given 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. Removing the redundant states reduces the number of states and yields a relatively more efficient CA.

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. Namely, whenever for all terms $t \in \mathbb{T}_\Sigma$ that satisfy $(E(s), N(s)) \models t$ it holds that $t[p] = t[q]$, for $\{p, q\} := L_C(s)$, s is said to be ✓-redundant. Similarly, whenever we can show that all consistent terms satisfy $t[p] \neq t[q]$ then s is ✗-redundant. Redundant states can be removed from the automata without affecting the correctness of its evaluation in the following way.

A state s that is ✓-redundant can be *removed* by updating δ such that the incoming transition $\delta(r, a) = s$, for some $r \in S$ and $a \in \{✓, ✗\}$, is updated to $\delta(s, ✓)$. A similar transformation of δ can be applied for states that are ✗-redundant using $\delta(s, ✗)$. We can observe that such a removal results in a relatively more efficient CA and that the size of the CA is reduced by the number of states in the ✓-branch (or ✗-branch) respectively if states unreachable by the transition relation are removed. Next, we prove that removal does not influence the correctness of evaluation.

▶ **Lemma 19.** *Let $M = (S, \delta, L, s_0)$ be any CA that is well-formed. Then the resulting CA M' where a ✓-redundant or ✗-redundant state $v \in S$ is removed remains well-formed.*

Using Lemma 19 and the fact that removing redundant states does not change the labelling of any state we have shown that EVAL-CA(M, s_0, t) = EVAL-CA(M', s_0, t) for all t.

If we consider Figure 2 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 local 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. Additional redundancies follow from the ordering of positions. For example, a term can never be equal to any of its subterms. Defining this complete procedure is left as future work.

4.5 Worst-case Complexity

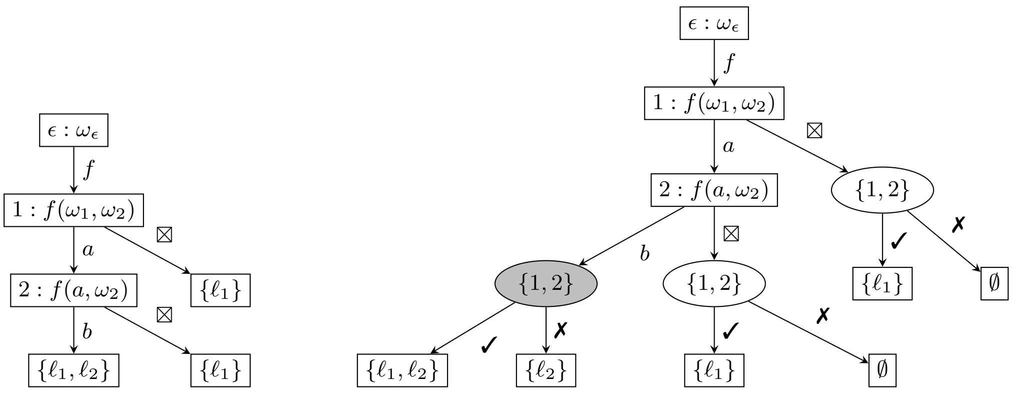
We establish several upper and lower bounds on the space and time complexity for consistency automata. The *maximum evaluation depth*, given by $\max_{t \in \mathbb{T}_\Sigma}(\mathrm{ED}(M,t))$, is the measurement for time complexity, where only the number of comparisons is counted. Finally, we only consider the time-optimal automaton M for the complexity analysis, which is the CA where the maximal evaluation depth is minimal from all possible selection functions.

For the time complexity of consistency automata we can show that each pair of positions is compared at most once. Let n be the number of unique position pairs in the given partitions, where each pair of positions is counted at most once. It can be shown that the worst-case time complexity of the consistency automata evaluation is tightly bounded by $\mathcal{O}(n)$ and its corresponding size is $\mathcal{O}(2^n)$. This follows essentially from the size of work for the first call to the construction procedure, which reduces in each recursive call. The given bounds are also tight as we can construct an example where the maximum evaluation depth requires exactly n comparisons.

5 Adaptive Non-linear Pattern Matching Automata

We have shown in Lemma 13 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 $(f(\omega_1,\omega_2),\{\{1,2\}\})$ and $(f(a,b)),\{\emptyset\})$. Now, the resulting APMA has a final state labelled with both patterns as shown in Figure 4a. 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.



(a) **(b)**

Figure 4 The resulting APMA shown on the left and the corresponding ANPMA with a grey ✗-redundant state on the right.

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 $f(a,b)$ is avoided. To enable these kind of efficiency improvements, we propose a combination of APMAs and CAs to obtain a matching automaton for non-linear patterns called *adaptive non-linear pattern matching automata*, abbreviated as ANPMAs. The result is an automaton that has three kinds of states; matching states of APMAs, consistency states of CAs and final states, and two transition functions; one for matching states and one for consistency states.

▶ **Definition 20.** *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 matching 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* $\delta_F : S_M \times \mathbb{F} \rightharpoonup S$ *and* $\delta_C : S_C \times \{✓, ✗\} \to S$*;*
- $L = L_M \uplus L_C \uplus L_F$ *is a state labelling function with* $L_M : S_M \to \mathbb{P}$*,* $L_C : S_C \to \mathbb{P}^2$ *and* $L_F : S_F \to 2^{\mathbb{T}}$*;*
- $s_0 \in S_M$ *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 $\textsc{Match}(M, s_0, t)$ below defines the evaluation of the ANPMA. It is essentially the combination of the evaluation functions for the APMA and CA depending on the current state.

$$\textsc{Match}\text{ANPMA}(M,t,s) = \begin{cases} L_M(s) & \text{if } s \in S_F \\ \textsc{Match}\text{ANPMA}(M,t,\delta_F(s,f)) & \text{if } s \in S_M \wedge \delta(s,f) \neq \bot \\ \textsc{Match}\text{ANPMA}(M,t,\delta_F(s,\boxtimes)) & \text{if } s \in S_M \wedge \delta(s,\boxtimes) \neq \bot \wedge \delta(s,f) = \bot \\ \emptyset & \text{if } s \in S_M \wedge \delta(s,\boxtimes) = \delta(s,f) = \bot \\ \textsc{Match}\text{ANPMA}(M,t,\delta_C(s,✓)) & \text{if } s \in S_C \wedge t[p] = t[q] \\ \textsc{Match}\text{ANPMA}(M,t,\delta_C(s,✗)) & \text{if } s \in S_C \wedge t[p] \neq t[q] \end{cases}$$

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

The construction algorithm of the ANPMA is defined in Algorithm 3. It combines the construction algorithm of APMAs (Algorithm 1) and the construction algorithm for CAs (Algorithm 2). 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 $\textsc{Select}$. 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.

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 $\textsc{Select}$ 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 matching state and the construction continues as in Algorithm 1. Otherwise, similar to Algorithm 2 two fresh states and two outgoing transition labelled with ✓ and ✗ are created, after which the parameters E and N are updated.

Algorithm 3 Given a set of patterns $\mathcal{L}$ and a renamed set of patterns $\mathcal{L}_r$, 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 $\mathsf{pref} = \omega_\epsilon$, and $E = N = \emptyset$.

```
procedure $\textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M, s, \mathsf{pref}, E, N)$
    $\mathcal{L}'_r := \{i : (\ell, P) \in \mathcal{L}_r \mid \ell$ unifies with $\mathsf{pref} \wedge \neg\exists C \in P : \exists\{p, q\} \in N : p, q \in C\}$
    $\mathsf{workF} := \mathcal{F}(\mathsf{pref})$
    $\mathsf{workC} := \{\{p, q\} \in \mathbb{P}^2 \mid \{p, q\} \subseteq\in P_i \wedge (i : \ell, i : P_i) \in \mathcal{L}'_r \wedge$ $\mathsf{pref}[p]$ and $\mathsf{pref}[q]$ are defined$\} \setminus E$
    if $(\mathsf{workF} = \emptyset$ and $\mathsf{workC} = \emptyset)$ or $(\mathcal{L}'_r = \emptyset)$ then
        $M := M[S_F := S_F \cup \{s\}, L := L[s \mapsto \{i : \ell \in \mathcal{L} \mid i : \ell' \in \mathcal{L}'\}]]$
    else
        $\mathsf{next} := \textsc{Select}(\mathsf{workF}, \mathsf{workC})$
        if $\mathsf{next} = \mathsf{pos}$ for some position $\mathsf{pos}$ then
            $M := M[S_M := (S_M \cup \{s\}), L_M := L_M[s \mapsto \mathsf{pos}]]$
            $F := \{f \in \mathbb{F} \mid \exists(i : (\ell, P)) \in \mathcal{L}'_r : \mathsf{head}(\ell[\mathsf{pos}]) = f\}$
            for $f \in F$ do
                $M := M[\delta := \delta[(s, f) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
                $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M, s', \mathsf{pref}[\mathsf{pos}/f(\omega_{\mathsf{pos}.1}, \ldots, \omega_{\mathsf{pos}.\mathsf{ar}(f)})], E, N)$
            if $\exists(i : (\ell, P)) \in \mathcal{L}'_r : \exists \mathsf{pos}' \leq \mathsf{pos} : \mathsf{head}(\ell[\mathsf{pos}']) \in \mathbb{V}$ then
                $M := M[\delta := \delta[(s, \boxtimes) \mapsto s']]$ where $s'$ is a fresh unbranded state w.r.t. $M$
                $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M, s', \mathsf{pref}[\mathsf{pos}/\boxtimes], E, N)$
        else if $\mathsf{next} = \{p, q\}$ for some pair $\{p, q\} \in \mathbb{P}^2$ then
            $M := M[S_C := (S_C \cup \{s\}), L_C := L_C[s \mapsto \{p, q\}]]$
            $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M[\delta_C := \delta_C[(s, ✓) \mapsto s']], s', \mathsf{pref}, E \cup \{\{p, q\}\}, N)$
              where $s'$ is an unbranded state w.r.t. $M$.
            $M := \textsc{ConstructANPMA}(\mathcal{L}, \mathcal{L}_r, \textsc{Select}, M[\delta_C := \delta_C[(s, ✗) \mapsto s']], s', \mathsf{pref}, E, N \cup \{\{p, q\}\})$
              where $s'$ is an unbranded state w.r.t. $M$.
    return $M$
```

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 9 and Theorem 17 and the proofs can be found in the appendix.

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

We extend the auxiliary definitions for APMA as follows. A *path* to s_n is a sequence with both types of labels $(s_0, a_0), \ldots, (s_{n-1}, a_{n-1}) \in S \times (\mathbb{F}_{\boxtimes} \uplus \{✓, ✗\})$ 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 $\mathsf{path}(s)$ such that $L(s_i).i = p$ for some $1 \leq i \leq \mathsf{ar}(f_i)$ 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 matching states in $\mathsf{path}(s)$ 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 21.** *The procedure* $\textsc{ConstructANPMA}(\mathcal{L}_r, P, \textsc{Select}, (\emptyset, \emptyset, \emptyset, s_0), s_0, \omega_\epsilon, \emptyset, \emptyset)$ *terminates and yields a well-formed ANPMA.*

Let $M = (S, \delta, L, s_0)$ be the ANPMA resulting from $\textsc{ConstructANPMA}(\mathcal{L}, \textsc{Select})$. 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 22.** *For all* $s \in S$ *such that* $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s)$ *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* $L(s_f) = \mathcal{L}_t$*.*

▶ **Lemma 23.** *If $\mathcal{L}_t = \emptyset$ then $\textsc{MatchANPMA}(M, t, s_0) = \emptyset$.*

▶ **Theorem 24.** *Then $\lambda t.\textsc{MatchANPMA}(M, t, s_0)$ is a matching function for $\mathcal{L}$.*

Proof. If $\mathcal{L}_t$ is empty then by Lemma 23 we get that $\textsc{MatchANPMA}(M, t, s_0) = \emptyset = \mathcal{L}_t$ as required. Otherwise, we have that $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s_f)$ for some final state s_f. Then by the definition of $\textsc{MatchANPMA}$ and Lemma 22 we conclude $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s_f) = L(s_f) = \mathcal{L}_t$. ◀

5.2 Strategy

The notion of ✓-redundancy (and ✗-redundancy) that we defined for CA can be easily extended to ANPMA. However, we can even identify more redundant states by considering the prefix for a given state s. Namely, for a state s labelled with a pair of positions $\{p, q\}$, given by $L_C(s)$, we can observe that s is ✗-redundant whenever $\mathsf{pref}[p]$ does not unify with $\mathsf{pref}[q]$, because if they do not unify then they can not be equal. Consider the patterns $\ell_1 : f(x, x)$ and $\ell_2 : f(a, b)$ again, we show the resulting ANPMA in Figure 4b.

6 Conclusion and Future Work

In this paper, we presented a formal proof for the correctness of APMAs. Furthermore, we introduced CAs 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.

ANPMAs 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. For the moment it is still unclear how to detect which states are redundant. An interesting direction for future research is to optimise the ANPMA construction algorithm that creates an optimal ANPMA on the fly.

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 any 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 APMAs some selection functions have already been studied in [8].

Thirdly it would be interesting to implement this approach. This work is a theoretical approach to ultimately obtain micro-optimisations in for example term rewriting. Many formalisms do not support non-linear patterns and as discussed in the introduction, many solutions to the pattern matching problem do not support it. It would be interesting to find out in practise whether exploiting $\mathcal{O}(1)$ term equality checking is worth the extra overhead that the ANPMA approach carries with it.

References

1 L. Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217. ACM, 1984.

2 J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993. `doi:10.1007/BF00881866`.

3 Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP*, pages 26–37. ACM, 2001.

4 Albert Gräf. Left-to-right tree pattern matching. In *RTA*, volume 488 of *LNCS*, pages 323–334. Springer, 1991.

5 P. Graf. Substitution tree indexing. In J. Hsiang, editor, *Rewriting Techniques and Applications*, pages 117–131, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

6 W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, October 1992. `doi:10.1007/BF00245458`.

7 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. `doi:10.1016/B978-044450813-3/50028-X`.

8 R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal of Computing*, 24(6):1207–1234, 1995. `doi:10.1137/S0097539793246252`.

9 M. van Weerdenburg. An account of implementing applicative term rewriting. *Electronic Notes in Theoretical Computer Science*, 174(10):139–155, 2007.

10 A. Voronkov. The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

A Proof of Lemma 4

Proof. The set $\mathcal{L}$ is finite, so all non recursive statements terminate. The for loop in particular treats finitely many function symbols from F. Finally, we show that the prefixes of the recursive calls are ordered by the matching ordering $<$. The algorithm CONSTRUCT realises that the prefix is only defined for defined positions of patterns in $\mathcal{L}$. Hence this ordering is well-founded on the recursive calls and the construction terminates.

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 CONSTRUCT($\mathcal{L}$, SELECT, M, s, 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. ◀

B Proof of Lemma 5

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 pref(s) is ground for final states s because the construction only creates final states if pref(s) has no variables.

c) By construction we have $L(s) = \{\ell \in \mathcal{L} \mid \ell$ unifies with pref(s)$\}$. Since pref(s) is ground we have that for all $\ell \in L(s)$ that $\ell \leq$ pref(s).

d) Let $\ell \in \mathcal{L}$. The following invariant holds for the construction algorithm: for all matching states s', if $\ell \in \mathcal{L}(s')$ then there is a pair (s'', f) such that $\delta(s', f) = s''$ and $\ell \in \mathcal{L}(s'')$. ◀

C Proof of Lemma 6

Proof. By induction on the length of $\mathsf{path}(s)$. If there are no pairs in $\mathsf{path}(s)$ then it must be that $s = s_0$. For the initial state we have $\mathcal{L}_t \subseteq \mathcal{L} = \mathcal{L}(s_0) = \mathcal{L}(s)$, so the base case holds.

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

- If $f \in \mathbb{F}$ then $\mathsf{pref}(s') = \mathsf{pref}(s)[p/f(\omega_{p.1}, \ldots, \omega_{p.\mathsf{ar}(f)})]$. By definition of $\textsc{Match}$ we know that $\mathsf{head}(t[p]) = f$.
  Let $\ell \in \mathcal{L}_t$. We show that ℓ unifies with $\mathsf{pref}(s')$. We know that $\ell \leq t$ by assumption. From the induction hypothesis it follows that ℓ unifies with $\mathsf{pref}(s)$. So there is a term u such that $\ell \leq u$ and $\mathsf{pref}(s) \leq u$. Then we distinguish two cases.
  - If $\ell[p']$ is a variable for some $p' \sqsubseteq p$ then ℓ unifies with $\mathsf{pref}(s')$.
  - If $\mathsf{head}(\ell[p])$ is a function symbol then by $\ell \leq t$ it must be that $\mathsf{head}(\ell[p]) = f$, so ℓ unifies with $\mathsf{pref}(s')$.
- If $f = \boxtimes$ then $\mathsf{pref}(s') = \mathsf{pref}(s)[p/\boxtimes]$. By definition of $\textsc{Match}$ we know that $\delta(s, \mathsf{head}(t[p]))$ is undefined.
  From the construction algorithm we then know that there is no pattern $\ell \in \mathcal{L}(s)$ such that $\mathsf{head}(\ell[p]) \in \mathbb{F}$ 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 $\mathsf{pref}(s)$. We show that ℓ unifies with $\mathsf{pref}(s')$ by showing that $\ell[p'] = \omega_{p'}$ for some position $p' \sqsubseteq p$.
  - Suppose that $\ell[p]$ exists. Since $\ell \leq t$ and $\mathsf{head}(t[p]) \neq \mathsf{head}(\ell[p])$ 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 $\mathsf{head}(\ell[p']) = f$ for some function symbol f. Then it must be that $\mathsf{head}(\mathsf{pref}(s)[p']) = f$ by the induction hypothesis. However, $\mathsf{pref}(s)[p]$ 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 $\ell[p'] = \omega_{p'}$. ◀

D Proof of Lemma 7

Proof.

a) We show that $\textsc{Match}(M, t, s_0) \neq L(s)$ 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 2 it follows that there is a position p and a function symbol $f \in \mathbb{F}$ such that $\mathsf{head}(\ell[p]) = f$ and $\mathsf{head}(t[p]) \neq f$. By Lemma 5 it must be that $\mathsf{head}(\mathsf{pref}(s)[p]) = f$, by which there must be a pair $(s_i, f) \in \mathsf{path}(s)$. Since $\textsc{Match}$ is a function we have $\textsc{Match}(M, t, s_0) = \textsc{Match}(M, t, s_i) = \textsc{Match}(M, t, s_f)$. However, by definition of $\textsc{Match}$ we know that $\mathsf{head}(t[p]) = f$, which contradicts the assumption that $l \in L(s_f)$.

b) Let $\ell \in \mathcal{L}_t$. We prove that for all s such that $\textsc{Match}(M, s_0, t) = \textsc{Match}(M, s, t)$, we have that $\delta(s, \mathsf{head}(t[L(s)]))$ or $\delta(s, \boxtimes)$ is defined.
Suppose that $\textsc{Match}(M, s_0, t) = \textsc{Match}(M, s, t)$. From Lemma 6 it follows that $\ell \in \mathcal{L}(s)$. If $\mathsf{head}(\ell[L(s)]) = f$ 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 $\mathsf{head}(\ell[L(s)])$ 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 $\textsc{Match}$ we then have that $\textsc{Match}(M, s_0, t)$ cannot yield the empty set, so it must terminate in a final state. ◀

E Proof of Lemma 8

Proof. Since $L(s_f) = \mathcal{L}(s_f)$ we know that $\mathcal{L}_t \subseteq L(s_f)$ by Lemma 6. It only remains show that $L(s_f) \subseteq \mathcal{L}_t$. Since s_f is a final state we have that $\mathsf{pref}(s_f)$ is ground; therefore $L(s_f) = \{\ell \in \mathcal{L} \mid \ell \leq \mathsf{pref}(s_f)\}$. Suppose for a contradiction that there is some $\ell \leq \mathsf{pref}(s_f)$ such that $\ell \not\leq t$. Then there is a position p such that $\mathsf{head}(\ell[p]) \in \mathbb{F}$ and $\mathsf{head}(t[p]) \neq \mathsf{head}(\ell[p])$. We have $\mathsf{head}(\ell[p]) = \mathsf{head}(\mathsf{pref}(s_f)[p])$ by assumption. So, there is a pair (s_i, f_i) in $\mathsf{path}(s_f)$ such that $L(s_i) = p$. By definition of Match we then have $\mathsf{head}(t[p]) = f_i = \mathsf{head}(\ell[p])$, a contradiction. ◀

F Proof of Lemma 12

Proof. We can show by induction on positions that $t =_\omega \mathsf{rename}_1(t, \epsilon)$ to prove the first statement. For the second statement let $p \in \mathcal{F}(t)$. First, we can show that $t'[p] = \mathsf{rename}_1(t[p], p)$ 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 $\{\{p \in \mathbb{P} \mid t[p] = x\} \mid x \in \mathsf{vars}(t)\}$ 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 $\mathsf{vars}(t)$. 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. ◀

G Proof of Lemma 15

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. ◀

H Proof of Lemma 16

Proof. The recursive calls form an *evaluation series* $(s_0, a_0), \ldots, (s_n, a_n)$ for $s_i \in S$ and $a_i \in \{✓, ✗\}$ for $0 \leq i < n$ such that $\textsc{eval-ca}(M, s_i, t) = \textsc{eval-ca}(M, s_{i+1}, t)$ 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 the statement holds for $\textsc{eval-ca}(M,t,s_0) = \textsc{eval-ca}(M,t,s)$. Suppose that $\textsc{eval-ca}(M,t,s) = \textsc{eval-ca}(M,t,s')$ where $s' = \delta(s,a)$ for $a \in \{✓,✗\}$ 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 $\delta(s,✓)$, 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. ◀

I Proof of Lemma 19

Proof. The recursive calls form an *evaluation series* $(s_0,a_0),\ldots,(s_n,a_n)$ for $s_i \in S$ and $a_i \in \{✓,✗\}$ for $0 \leq i < n$ such that $\textsc{eval-ca}(M,s_i,t) = \textsc{eval-ca}(M,s_{i+1},t)$ 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 $(s_0,a_0),\ldots,(s_k,a_k) \in (S \times \{✓,✗\})$ of $\textsc{eval-ca}(M,s_0,t)$, 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 $a \in \{✓,✗\}$, and let t be an arbitrary term. Note that the initial state can not be removed by this procedure. Let $\{p,q\}$ be the value of $L_C(v)$ then there are two cases to consider:

- v is ✓-redundant. It follows that $t[p] = t[q]$ for $\{p,q\} := L_C(s)$. All sequences such that v occurs in it must contain exactly the pair $(v,✓)$ by definition of ✓-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.
- s is ✗-redundant. Similarly, with the observation that $(E(v), N(v) \cup \{\{p,q\}\}) \models t$. ◀

J Proof for Lemma 21

Proof. We only show that the recursion terminates. The rest is similar to the proof for Lemma 4, with the additional observation that positions in P are only chosen when they are defined in the prefix. Given the parameters $\mathsf{pref}_1, E_1, N_1$ and $\mathsf{pref}_2, E_2, N_2$ we can fix the ordering:

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

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 to this ordering; therefore the recursion terminates. ◀

K Proof for Lemma 22

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

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 $\textsc{MatchANPMA}(A, t, s_0) = \textsc{MatchANPMA}(A, t, s)$. Suppose $\textsc{MatchANPMA}(A, t, s) = \textsc{MatchANPMA}(A, t, s')$ for some $s' = \delta(s, x)$ such that $x \in (\mathbb{F}_{\boxtimes} \uplus \{\checkmark, \text{✗}\})$. 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 $\mathsf{pref}(s') = \mathsf{pref}(s)$.
  - 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 $i : l \in \mathcal{L}(s)$ such that $i : l \notin \mathcal{L}(s')$. From $\mathsf{pref}(s') = \mathsf{pref}(s)$ it follows that for $i : (l', P) \in \mathcal{L}_r$ it holds that P is not consistent w.r.t. t by observation that positions $\{p, q\} \subseteq\in P$ are included in N and $t[p] \neq t[q]$. Therefore, by Lemma 12 it holds that $i : l \notin \mathcal{L}_t$.
- $s \in S_M$. It hold 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 $\mathsf{pref}(s')$. Then the same arguments as given in Lemma 6 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 $\mathsf{workC} = \emptyset$ and $\mathsf{workF} = \emptyset$. Suppose for a contradiction that there is some $i : l \in L(s_f)$ such that $i : l \notin \mathcal{L}_t$. It follows that $l \not\leq t$, which means that for $i : (l', P) \in \mathcal{L}_r$ that $l' \not\leq t$ or t is not consistent w.r.t. P by Lemma 12. We show that both cases lead to a contradiction:
  - Case $l' \not\leq t$. This follows essentially from the same observations as Lemma 8.
  - Case t is not consistent w.r.t. P. From the fact that $\mathsf{pref}(s_f)$ unifies with t and that it is a ground term due to $\mathsf{workF} = \emptyset$ it follows that for all p, q such that $\{p, q\} \subseteq\in P_i$ they are defined in $\mathsf{pref}(s)$ 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. ◀

L Proof for Lemma 23

Proof. We show that $\textsc{Match}(M, t, s_0) \neq L(s)$ for all final states s for which for $L(s) \neq \emptyset$. Let s_f be an arbitrary final state such that $L(s) \neq \emptyset$ and pick some pattern $i : \ell \in L(s_f)$. By assumption $\ell \not\leq t$ and by Lemma 12 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 2 it follows that there is a position p and a function symbol $f \in \mathbb{F}$ such that $\mathsf{head}(\ell[p]) = f$ and $\mathsf{head}(t[p]) \neq f$. By Lemma 5 it must be that $\mathsf{head}(\mathsf{pref}(s)[p]) = f$, by which there must be a pair $(s_i, f) \in \mathsf{path}(s)$. Since $\textsc{MatchANPMA}$ is a function we again have that $\textsc{MatchANPMA}(M, t, s_0) = \textsc{MatchANPMA}(M, t, s_i) = \textsc{MatchANPMA}(M, t, s_f)$. However, by its definition we know that $\mathsf{head}(t[p]) = f$, which contradicts the assumption that $i : l \in L(s_f)$.
- If t is not consistent w.r.t. P. By Lemma 22 we know that $(E(s_f), N(s_f)) \models t$ and for all pairs $\{p, q\} \subseteq\in P$ it holds that $\{p, q\} \in E$ for workC to become empty, because all positions of pattern l are defined in the prefix $\mathsf{pref}(s_f)$. As such t must be consistent w.r.t. P, which contradicts the assumption that $i : l \in L(s_f)$. ◀