Verifying Higher-Order Functional Programs with Tree Automata

Thomas Genet, Timothée Haudebourg, Thomas Jensen

Verifying Higher-Order Functional Programs with Tree Automata: Extended Version

Thomas Genet, Timothée Haudebourg, and Thomas Jensen

Irisa, Rennes, France

Abstract. This paper describes a fully automatic technique for verifying properties of higher-order functional programs. Functional programs are modelled with term rewriting systems and tree automata are used to model reachable program states. From a tree automaton representing the initial state, it is possible to define a completion algorithm on tree automata for approximating the output set of the program to verify. We define a subclass of higher-order functional programs for which the completion is guaranteed to terminate. Furthermore, since precision of the completion is fixed by a set of equations, we also propose an algorithm to automatically generate sets of equations by iterative refinement. We present some experiments showing that the resulting verification technique is complementary and extends several state-of-the-art model-checking approaches for higher-order functional programs.

1 Introduction

Higher-order functions are ubiquitous in modern programming languages such as Java, Scala or JavaScript, not to mention Haskell and Caml. With this widespread use of higher-order functions comes the need for reasoning about the correctness of programs that employ them. To this end, the correctness-minded software engineer can opt for proving properties interactively with the help of a proof assistant such as Coq [8] or Isabelle/HOL [**?**], or write a specification in a formalism such as Liquid Types [17] or Bounded Refinement Types [22,21] and ask an SMT solver whether it can prove the verification conditions generated from this specification. This approach requires expertise of the formal method used and both the proof construction and the annotation phase can be time consuming.

Another approach is based on *fully automated* verification tools, where the proof is carried out automatically without annotations or intermediate lemmas. This approach is accessible to a larger class of programmers but applies to a more restricted class of program properties which will provide approximate information about the program's behaviour.The static analysis of higher-order functions has been studied extensively in the setting of abstract interpretation. [**?**,**?**,**?**,**?**]. Abstract interpretation for first-order imperative program [3,2] has shown to scale, but the technique has problems with representing the immense set of possible states in higher-order programs. More recently, the breakthrough results of Ong *et. al* [16,13] and Kobayashi [15,14,19] show that combining abstraction

with model checking techniques can be used with success to analyse higher-order functions automatically. This approach also relies on abstraction for computing over-approximations of the set of reachable states, on which safety properties can then be verified.

In this paper, we pursue the approach to higher-order functional verification initiated by Ong and Kobayashi and present a formal verification technique based on Tree Automata Completion (TAC) [**?**]. This method is able to check a class of properties on programs, called *regular properties*, in a fully automatic manner. In our approach, a program is represented as a term rewriting system $\mathcal{R}$ and the set of (possibly infinite) inputs to this program as a tree automaton $\mathcal{A}$. The TAC algorithm computes a new automaton $\mathcal{A}^*$, by *completing* $\mathcal{A}$ with all terms reachable from $\mathcal{A}$ by $\mathcal{R}$-rewriting. This automaton representation of the *reachable terms* contains all intermediate states as well as the final output of the program. Checking correctness properties of the program is then reduced to checking properties of its corresponding automata.

In the following example the TRS $\mathcal{R}$ defines the function *filter* that returns its input list with every element that does not satisfy a boolean function p (given as parameter) filtered out. Higher-order programs treat functions as "first-class citizen", so functions are treated as any other constructor of arity 0 (such as *nil*). Function application is written by a special *application* symbol @ to denote function application, so $@(f, x)$ means "x applied to f". This example also defines two predicates *even* and *odd* on Peano's natural numbers.

Example 1. Let the TRS $\mathcal{R}$ be defined by the six rules:

$$@(@(\mathit{filter}, \underline{p}), \mathit{cons}(\underline{x}, \underline{l})) \to \textbf{if } @(\underline{p}, \underline{x}) \textbf{ then } \mathit{cons}(\underline{x}, @(@(\mathit{filter}, \underline{p}), \underline{l})) \textbf{ else } @(@(\mathit{filter}, \underline{p}), \underline{l})$$

$$@(@(\mathit{filter}, \underline{p}), \mathit{nil}) \to \mathit{nil}$$

$$@(\mathit{even}, 0) \to \mathit{true} \qquad @(\mathit{even}, s(\underline{x})) \to @(\mathit{odd}, \underline{x})$$

$$@(\mathit{odd}, 0) \to \mathit{false} \qquad @(\mathit{odd}, s(\underline{x})) \to @(\mathit{even}, \underline{x})$$

We want to check that for all lists l of natural numbers $@(@(\mathit{filter}, \mathit{odd}), l)$ filters out all even numbers. One way to do this is to write a higher-order predicate, *exists*, and check that there exists no even number in the resulting list, i.e. that $@(@(\mathit{exists}, \mathit{even}), @(@(\mathit{filter}, \mathit{odd}), l))$ always rewrites to false. Let $\mathcal{A}$ be the tree automaton recognizing terms of form $@(@(\mathit{exists}, \mathit{even}), @(@(\mathit{filter}, \mathit{odd}), l))$ where l is any list of natural numbers. The completion algorithm computes an automaton $\mathcal{A}^*$ recognizing every term reachable from $L(\mathcal{A})$ using $\mathcal{R}$ where we add the definition of the *exists* function. Formally,

$$L(\mathcal{A}^*) = \mathcal{R}^*(L(\mathcal{A})) = \{t \mid \exists s \in L(\mathcal{A}), s \to_{\mathcal{R}}^* t\}$$

To prove the expected property, it suffices to check that *true* is not reachable, i.e. $\mathit{true} \notin L(\mathcal{A}^*)$. Note that even in the case when $\mathcal{A}^*$ is an over-approximation ($L(\mathcal{A}^*) \supseteq \mathcal{R}^*(L(\mathcal{A}))$), $\mathit{true} \notin L(\mathcal{A}^*)$ implies that the same property holds. We

denote by *regular properties* the family of properties we can show on a program using this technique. Roughly, regular properties do not count symbols in terms, nor relate subterm heights. A non regular property is, for instance, the fact that the length of $@(@(\mathit{filter}, \mathit{odd}), l))$ is smaller or equal to the length of l. In our setting, if we encode the function *length* and $\leq$ in $\mathcal{R}$, in all regular over-approximations of terms reachable from $@(@(\leq, @(\mathit{length}, @(@(\mathit{filter}, \mathit{odd}), l))), @(\mathit{length}, l))$ the terms $true$ and $false$ will be both reachable. The class of properties that is nicely complements properties covered by other existing tools. We identify a class of higher-order functional programs for which the over-approximation of reachable states always succeeds. This class is quite comprehensive and includes standard usage of higher-order features in functional programming languages.

Termination of the tree automata completion algorithm is not ensured [10] in general. For instance when the target language $\mathcal{R}^*(L(\mathcal{A}))$ is not regular, it cannot be represented as a tree automaton, so the iterative computation $\mathcal{A}^*$ will grow the automaton without ever converging. In this case, the user can provide a set of *equations* that will force termination by introducing *approximations*: $\mathcal{R}^*(L(\mathcal{A})) \subseteq L(\mathcal{A}^*)$. Equations make TAC powerful enough to verify first-order functional programs. However, state-of-the-art TAC has two short-comings. (i) Equations must be given by the user, which goes against the goal of full automation, and (ii) even with equations, *termination is not guaranteed* in the case of higher-order programs. In this paper we propose a solution to these short-comings in the following way:

- We state and prove a general termination theorem for the Tree Automata Completion algorithm (Section 3);
- From the conditions of the theorem we characterise a subclass of higher-order functional programs for which the completion algorithm terminates (Section 4). This class is compatible with programmers' usage of higher-order features in functional programming languages.
- We define an algorithm that is able to automatically generate equations for enforcing convergence, thus avoiding any user intervention (Section 5).

The rest of this paper is organized as follow: First we give a description of the completion algorithm (Section 2), how it works, and how to use equations to ensure termination. Then we present our contribution as mentioned above (Sections 3 to 5). We illustrate the strengths and the weakness of our verification technique with some experiments (Section 6). Finally we discuss related work (Section 7) and future work (Section 8). Section 9 concludes the paper.

2 Term rewriting and tree automata

In this section we present background material on approximating term rewriting systems with tree automata.

2.1 Terms and Rewriting Systems

An alphabet $\mathcal{F}$ is a finite set of symbols, with an arity function $ar : \mathcal{F} \to \mathbb{N}$. Symbols represent constructors such as *nil* or *cons*, or functions such as *filter*,

etc. For simplicity, we also write $f \in \mathcal{F}^n$ when $f \in \mathcal{F}$ and $ar(f) = n$. For instance, $cons \in \mathcal{F}^2$ and $nil \in \mathcal{F}^0$. An alphabet $\mathcal{F}$ and finite set of variables $\mathcal{X}$ together induces a set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that:

$$\underline{x} \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \Leftarrow \underline{x} \in \mathcal{X}$$
$$f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \Leftarrow f \in \mathcal{F}^n \text{ and } t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$$

Variables are underlined when $\mathcal{X}$ is not defined explicitly. A set of term is called a *language*. A term t is said *linear* if the multiplicity of each variable in t is at most 1. A term is *closed* if it contains no variables. The set of all closed terms is written $\mathcal{T}(\mathcal{F})$. A *position* in a term t is a word over $\mathbb{N}$ pointing to a *subterm* of t. $Pos(t)$ is the set of all positions in t, one for each subterm of t. $Pos(t)$ is inductively defined by:

$$Pos(\underline{x}) = \{\lambda\}$$
$$Pos(f(t_1, \ldots, t_n)) = \{\lambda\} \cup \{i.p \mid 1 \leq i \leq n \wedge p \in Pos(t_i)\}$$

where "." in $i.p$ is the *concatenation* operator. For $p \in Pos(t)$, we write $t|_p$ for the subterm of t at position p, and $t[s]_p$ for the term t where the subterm at position p has been replaced by s. *E.g.*, let $\mathcal{F} = \{cons, nil\}$, $t = cons(\underline{x}, cons(\underline{y}, nil))$ and p the position $p = 2.1$. Then, $t|_\lambda = t$, $t|_p = \underline{y}$ and $t[nil]_p = cons(\underline{x}, cons(nil, nil))$. We write $s \trianglerighteq t$ if t is a subterm of s and $s \triangleright t$ if it is a subterm and $s \neq t$. We note $\mathcal{L}_\trianglerighteq$ the language $\mathcal{L}$ closed by subterms.

A *substitution* σ is an application of $\mathcal{X} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$, mapping a variable to a term. We tacitly extend it to the endomorphism $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{X}) \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$ where $t\sigma$ is the result of the application of the term t to the substitution σ. A context $C[\,]$ is a term of $\mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{X})$ where $\Box$ is a special "hole" symbol. In this paper we consider contexts containing a unique symbol $\Box$. We note $C[t] = C[\,]\sigma$ with $\sigma = \{\Box \mapsto t\}$.

If terms can model the expressions in a functional program, *term rewriting systems* [1] are a convenient way to define the program itself and its semantics. A rewriting system is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is an alphabet and $\mathcal{R}$ a set of rewriting rules of the form $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$. A TRS can be seen as a set of rules, each of them defining one step of computation. We write $\mathcal{R}$ a rewriting system $\langle \mathcal{F}, \mathcal{R} \rangle$ if there is no ambiguity on $\mathcal{F}$. A rewriting rule $l \to r$ is said to be left-linear if the term l is linear. Example 1 shows a TRS representing a functional program, where each rule is left-linear. In that case we say that the TRS $\mathcal{R}$ is left-linear.

A rewriting system $\mathcal{R}$ induces a rewriting relation $\to_\mathcal{R}$ where for alls $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \to_\mathcal{R} t$ if it exists a rule $l \to r \in \mathcal{R}$, a position $p \in Pos(s)$ and a substitution σ such that $l\sigma = s|_p$ and $t = s[r\sigma]_p$. The reflexive-transitive closure of $\to_\mathcal{R}$ is written $\to^*_\mathcal{R}$. The rewriting system introduced in the previous example also derives a rewriting relation $\to_\mathcal{R}$ where

$$@(@(filter, odd), cons(0, cons(s(0), nil))) \to^*_\mathcal{R} cons(s(0), nil)$$

No rule can be applied to the last term $cons(s(0), nil)$. It is *irreducible*, and is the result of the function call. $IRR(\mathcal{R})$ is the language of irreducible terms of $\mathcal{R}$.

2.2 Tree Automata Completion

Tree automata [7] are a convenient way to represent regular sets of terms. A tree automaton is a quadruple $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ where $\mathcal{F}$ is an alphabet, $\mathcal{Q}$ is a finite set of states, $\mathcal{Q}_f$ is a set of *final states*, and Δ is a rewriting system on $\mathcal{F} \cup \mathcal{Q}$. Each rule in Δ is of the form $l \to q$ where $q \in \mathcal{Q}$ and l is either a state ($\in \mathcal{Q}$), or a *configuration* of the form $f(q_1, \ldots, q_n)$ with $f \in \mathcal{F}, q_1 \ldots q_n \in \mathcal{Q}$. A term t is *recognized* by a state $q \in \mathcal{Q}$ if $t \to_\Delta^* q$. We note $L(\mathcal{A}, q)$ the language of all terms recognized by q. t is recognized by $\mathcal{A}$ if it exists a final state $q \in \mathcal{Q}_f$ such that $t \in L(\mathcal{A}, q)$. In that case we note $t \in L(\mathcal{A})$. For example, the tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ with $\mathcal{F} = \{nil : 0, cons : 2, 0 : 0, s : 2\}$ and $\mathcal{Q}_f = \{q_{list}\}$ and $\Delta =$

$$\begin{array}{rr} 0 \to q_{\mathbb{N}} & nil \to q_{list} \\ s(q_{\mathbb{N}}) \to q_{\mathbb{N}} & cons(q_{\mathbb{N}}, q_{list}) \to q_{list} \end{array}$$

recognizes all lists of natural numbers.

A tree automaton $\mathcal{A}$ is *ϵ-free* if it contains no ϵ-transition (of the form $q \to q'$). We note $t \to_\Delta^{\not\epsilon *} q$ if t is recognized by q with no ϵ-transition. $\mathcal{A}$ is *deterministic* if for all terms t there is at most one state q such that $t \to_\Delta^* q$. $\mathcal{A}$ is *reduced* if for all states q there is at least one term t such that $t \to_\Delta^* q$.

The verification algorithm is based on *tree automata completion*. Given a program represented as a rewriting system $\mathcal{R}$, and its input represented as a tree automaton $\mathcal{A}$, the *tree automata completion algorithm* computes a new tree automaton $\mathcal{A}^*$ recognizing the set of all *reachable terms* starting from a term in $L(\mathcal{A})$. For a given $\mathcal{R}$, we write this $\mathcal{R}^*(L(\mathcal{A})) = \{t \mid \exists s \in L(\mathcal{A}), s \to_\mathcal{R}^* t\}$. This set includes all intermediate computations and, in particular, the *output* of the functional program.

It is in general impossible to compute exactly, so instead we shall over-approximate it by an automaton $\mathcal{A}^*$ such that $L(\mathcal{A}^*) \supseteq \mathcal{R}^*(L(\mathcal{A}))$. The approximation is performed using a set E of *equations* of the form $l = r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. E derives a relation $=_E$, the *smallest congruence* such that for all equation $l = r$ and substitution σ we have $l\sigma =_E r\sigma$

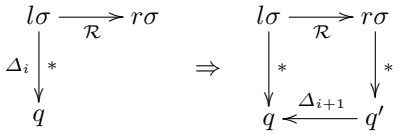
Let $\mathcal{R}$ be a left-linear TRS and $\mathcal{A}_0 = \langle F, Q, Q_f, \Delta_0 \rangle$ a tree automaton. Formally, when completing $\mathcal{A}_0$ by $\mathcal{R}$ and E the completion algorithm iteratively computes $\mathcal{A}^1_{\mathcal{R},E}, \mathcal{A}^2_{\mathcal{R},E}, \ldots$ such that for all $i \in \mathbb{N}$:

$$L(\mathcal{A}^i_{\mathcal{R},E}) \subseteq L(\mathcal{A}^{i+1}) \quad \text{and}$$
$$s \in L(\mathcal{A}^i_{\mathcal{R},E}) \;\Rightarrow\; s \to_\mathcal{R} t \;\Rightarrow\; t \in L(\mathcal{A}^{i+1}_{\mathcal{R},E})$$

It terminates when a fixpoint, noted $\mathcal{A}^*$, is reached. At each iteration, $\mathcal{A}^{i+1}_{\mathcal{R},E}$ is computed as $\mathcal{A}^{i+1}_{\mathcal{R},E} = \mathcal{S}_E(\mathcal{C}_\mathcal{R}(\mathcal{A}^i_{\mathcal{R},E}))$ where $\mathcal{C}_\mathcal{R}(\cdot)$ is one *completion step* using

$\mathcal{R}$ and $\mathcal{S}_E(\cdot)$ one *simplification step* using E. The completion step consists in finding and *completing* the *critical pairs* of $\mathcal{A}^i_{\mathcal{R},E}$.

Definition 1. *A critical pair is a triple $\langle l \to r, \sigma, q\rangle$ where $l \to r \in \mathcal{R}$, σ is a substitution, and $q \in \mathcal{Q}$ such that $l\sigma \to^*_\Delta q$.*



A critical pair signals a term $l\sigma$ recognized by $\mathcal{A}^i_{\mathcal{R},E}$ that can be rewritten by $\mathcal{R}$ into $r\sigma$. It is completed by adding the necessary states and transitions in $\mathcal{A}^{i+1}_{\mathcal{R},E}$ to have $r\sigma \in L(\mathcal{A}^{i+1}_{\mathcal{R},E})$. The current implementation adds a state q' such that $r\sigma \in L(\mathcal{A}^{i+1}_{\mathcal{R},E}, q')$ and the transition $q \to q'$. The simplification step merges the states q, q' such that there exists two terms $s =_E t$ and $s \to^{\not\epsilon *}_\Delta q$ and $t \to^{\not\epsilon *}_\Delta q'$. This results in an over-approximation of the language originally recognized by the automaton: $L(\mathcal{A}) \subseteq L(\mathcal{S}_E(\mathcal{A}))$.

3 Completion Termination

Given $\mathcal{R}$, $\mathcal{A}$ and E, the termination of the completion algorithm, even with the use of equations, is undecidable in general. In this section, we show that we can prove termination under the following conditions: if (i) $\mathcal{A}^k_{\mathcal{R},E}$ is reduced ϵ-free and deterministic (REFD) for all k; (ii) every term added to $\mathcal{A}$ during completion can be reduced into a term of a language $\mathcal{L}$; (iii) there is a finite number of normal forms of $\mathcal{L}$ w.r.t $\vec{E}$. Completion is known to preserve $\not\epsilon$-reduceness and $\not\epsilon$-determinism if $E \supseteq E_r \cup E_\mathcal{R}$ [10] where $E_\mathcal{R} = \{s = t \mid s \to t \in \mathcal{R}\}$ and $E_r = \{f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \mid f \in \mathcal{F}^n\}$. To ensure condition (i), we show in addition that in our verification settings, completion preserve REFD. The last condition is ensured by having $E \supseteq E^c_\mathcal{L}$ where $E^c_\mathcal{L}$ is a set of *contracting equations*.

Definition 2 (Contracting Equations). *Let $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$. A set of equations is contracting for $\mathcal{L}$, denoted by $E^c_\mathcal{L}$, if all equations of $E^c_\mathcal{L}$ are of the form $u = u|_p$ with u a linear term of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $p \neq \lambda$ and if the set of normal forms of $\mathcal{L}$ w.r.t the TRS $\vec{E^c_\mathcal{L}} = \{u \to u|_p \mid u = u|_p \in E^c_\mathcal{L}\}$ is finite.*

The contracting equations ensure that the completion algorithm will merge enough states during the simplification steps to terminate. First, we prove that it is possible to bound the number of states needed in $\mathcal{A}^*$ to recognize a language $\mathcal{L}$ by the number of normal forms of $\mathcal{L}$ w.r.t $\vec{E^c_\mathcal{L}}$ (Lemma 2). In our case $\mathcal{L}$ will be the set of output terms of the program. However, as we have seen, $\mathcal{A}^*$ does not only recognizes the output terms, and many more states could be added to the automaton for the recognition of the intermediate computation

terms. Our Theorem 1 states that with $E_{\mathcal{R}}$, the simplification steps will merge the states recognizing the intermediate computation with the states recognizing the outputs. If there is a finite amount of them (using Lemma 2), then $\mathcal{A}^*$ is finite.

This theorem only holds for REFD tree automata. Thus we first need to prove that completion preserves REFD. It is already known that completion preserves $\not\epsilon$-determinism and $\not\epsilon$-reduction [10]. Considering that an $\not\epsilon$-deterministic and $\not\epsilon$-reduced automaton that is ϵ-free is REFD by definition, our first lemma prove preservation of ϵ-freeness.

Lemma 1. *Let $\mathcal{A}$ be an ϵ-free tree automaton and E a set of equations. If $E \supseteq E_{\mathcal{R}}$ then $\mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}))$ is ϵ-free.*

Proof. $\mathcal{A}$ is ϵ-free so for each critical pair $\langle l \to r, \sigma, q\rangle$ we have $l\sigma \to^{\not\epsilon}_{\mathcal{A}} q$. The resolution of the critical pair does not add any ϵ-transition but $q \to q'$ where q' is a new state [10]. So in $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$, for every transition $q \to q'$ such that $q \neq q'$ there exists s, t such that $s \to^{\not\epsilon} q$, $t \to^{\not\epsilon} q'$ and $s \to_{\mathcal{R}} t$. Now since $E \supseteq E_{\mathcal{R}}$, we have $s =_E t$, and q and q' are merged in the simplified automaton $\mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}))$. The only epsilon transitions after the simplification steps are of the form $q \to q$ and can be removed without altering the recognized language.

Now let's prove that with a REFD tree-automaton $\mathcal{A}$, the number of states needed in $\mathcal{S}_E(\mathcal{A})$ to recognize a language $\mathcal{L}$ is bounded by the number of normal forms of $\mathcal{L}$ w.r.t $\vec{E^c_{\mathcal{L}}}$.

Lemma 2. *Let $\mathcal{A}$ be a REFD tree automaton, $\mathcal{L}$ a language and E a set of equations such that $E \supseteq E_r \cup E^c_{\mathcal{L}}$. Let's name G the set of normal forms of $\mathcal{L}$ w.r.t $\vec{E^c_{\mathcal{L}}}$. If G is finite then $\mathcal{S}_E(\mathcal{A})$ is a deterministic automaton where terms of $\mathcal{L}$ are recognized by no more states that terms in G.*

Proof. First we prove that for all terms $s \in \mathcal{L}$, if $s \to^*_{\mathcal{S}_E(\mathcal{A})} q$ then there exists $t \in G$ such that $t \to^*_{\mathcal{S}_E(\mathcal{A})} q$. If $s \in G$ then it is trivially true. If s is not in normal form w.r.t $\vec{E^c_{\mathcal{L}}}$ then there exists a subterm $t \in G$ of s such that $s \to^*_{\vec{E^c_{\mathcal{L}}}} t$. Since t is a subterm of s, there exists a state q' such that $t \to^*_{\mathcal{S}_E(\mathcal{A})} q'$. Thanks to the form of $E^c_{\mathcal{L}}$ it guarantees that $q = q'$ in the simplified automaton $\mathcal{S}_E(\mathcal{A})$. t is a normal form of G recognized by q. $\mathcal{S}_E(\mathcal{A})$ being ϵ-free deterministic 1, for all terms $t \in G$ there is at most one state q such that $t \to^*_{\mathcal{S}_E(\mathcal{A})} q$. Moreover we have just seen that every state recognizing a term of $\mathcal{L}$ recognizes a term of G. Hence there is no more state in $\mathcal{S}_E(\mathcal{A})$ than term in G. □

Now we state and prove the termination theorem, using $E \supseteq E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$. The proof idea is shown in Figure 1. Each cross represents a term recognized by the completed automaton. The simple arrows represent the rewriting relation $\mathcal{R}$ used to add new terms during completion, and the dashed arrows represent the contracting relation derived from $\vec{E^c_{\mathcal{L}}}$. All terms linked together with arrows are bound by the relation E (since E contains $E_{\mathcal{R}}$ and $E^c_{\mathcal{L}}$) and recognized by the

same states. G is the set of normal forms of $\mathcal{L}$ w.r.t $\vec{E^c_{\mathcal{L}}}$, so there cannot be more that $|G|$ states in $\mathcal{A}^*$. On this figure it means that the completed automaton contains only two states.

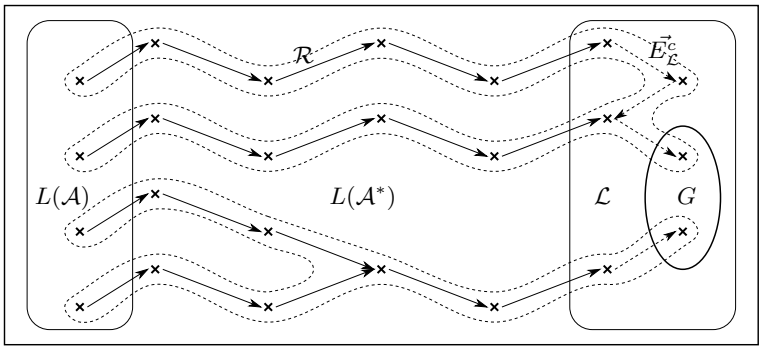


Fig. 1: Completion termination proof idea

Theorem 1. *Let $\mathcal{A}$ be a REFD tree automaton, $\mathcal{R}$ a left-linear TRS, E a set of equations and a language $\mathcal{L}$ closed by subterm such that for all $k \in \mathbb{N}$ and for all $s \in L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$, there exists $t \in \mathcal{L}$ s.t. $s \to^*_{\mathcal{R}} t$. If $E \supseteq E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$ then the completion of $\mathcal{A}$ by R and E terminates.*

Proof. Let $G \subseteq \mathcal{L}$ be the set of normal forms of $\mathcal{L}$ w.r.t $\vec{E^c_{\mathcal{L}}}$. By hypothesis, G is finite. The number of states recognizing terms of $\mathcal{L}$ is finite (Lemma 2), For all states q of $\mathcal{A}^k_{\mathcal{R},E}$ there exists a term $s \in L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ such that $s \to^{\not\epsilon *}_{\mathcal{A}^k_{\mathcal{R},E}} q$ ($\mathcal{A}$ is reduced). By hypothesis, there exists $t \in \mathcal{L}$ s.t. $s \to^*_{\mathcal{R}} t$. Since $E \supseteq E_{\mathcal{R}}$ we have $s =_E t$. And because $\mathcal{A}^k_{\mathcal{R},E}$ is simplified and ϵ-free, it means that there can be only one state recognizing s and t. Hence there is no more states recognizing terms of $L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ than states recognizing terms of $\mathcal{L}$. The number of states recognizing terms of $\mathcal{L}$ being finite, the number of states recognizing terms of $L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ is finite, for all k. In particular, the number of states recognizing terms of $L_{\trianglerighteq}(\mathcal{A}^*)$ is finite, and since for all states q, $L(\mathcal{A}^*, q) \subseteq L_{\trianglerighteq}(\mathcal{A}^*)$, the number of states in $\mathcal{A}^*$ is finite. □

4 Terminating Programs

By choosing $\mathcal{L} = \mathcal{T}(\mathcal{F})$ and providing a set of contracting equations $E^c_{\mathcal{T}(\mathcal{F})}$, the termination theorem above can be used to prove that the completion algorithm terminate on any functional program $\mathcal{R}$. If this works in theory, in practice we want to avoid to produce equations over the application symbol (such as $@(x, y) = y$). Contracting equations on applications can make sense in certain cases, for instance with idempotent functions ($@(sort, @(sort, x)) = @(sort, x)$).

However in most cases, such equations dramatically damage the precision of the completion algorithm.

We want to find a class of functional programs and a language $\mathcal{L}$ for which Theorem 1 applies with no contracting equations over @ in $E^c_{\mathcal{L}}$. Since such language $\mathcal{L}$ still has to have a finite number of normal forms w.r.t. $\vec{E^c_{\mathcal{L}}}$, it cannot include a term containing non bounded *stack* of applications. For instance, $\mathcal{L}$ cannot contain all the terms of the form $@(f,x), @(f,@(f,x)), @(f,@(f,@(f,x))$, etc. The @ stack must be bounded, even if the applications symbols are interleaved with other symbols (e.g. $@(f,s(@(f,s(@(f,s(x))))))$). We define $\mathcal{B}^n$ the set of term where such stack size is bounded by n.

Definition 3. *For a given alphabet $\mathcal{F} = \mathcal{C} \cup \{@\}$, $\mathcal{B}^n$ is the set of terms where every application depth is bounded by n. It is the smallest set defined by:*

$$\begin{aligned} f \in \mathcal{B}^0 &\Leftarrow f \in \mathcal{C}^0 \\ f(t_1,\ldots,t_n) \in \mathcal{B}^i &\Leftarrow f \in \mathcal{C}^n \wedge t_1 \ldots t_n \in \mathcal{B}^i \\ @(t_1,t_2) \in \mathcal{B}^{i+1} &\Leftarrow t_1, t_2 \in \mathcal{B}^i \\ t \in \mathcal{B}^{i+1} &\Leftarrow t \in \mathcal{B}^i \end{aligned}$$

In Section 5 we show how to produce E^c such that $\mathcal{B}^n \cap IRR(\mathcal{R})$ has a finite number of normal forms w.r.t. $\vec{E^c}$ with no equations on @. Now if we can show that for all k, for all term $t \in L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ there exists $s \in \mathcal{B}^n \cap IRR(\mathcal{R})$ s.t. $t \to^*_{\mathcal{R}} s$, Theorem 1 can be instanciated with $\mathcal{L} = \mathcal{B}^n \cap IRR(\mathcal{R})$. This is not true in general. In the following, we define a set $\mathcal{K}^n \subseteq \mathcal{T}(\mathcal{F})$ and a class of TRS $\mathcal{K}$ such that (i) $\mathcal{K}^n \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^n$ and (ii) $L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E}) \subseteq \mathcal{K}^n_{\trianglerighteq}$. If $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n_{\trianglerighteq}$ we can instanciate Theorem 1 with $\mathcal{L} = \mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R})$ and prove termination. Here $\mathcal{K}$ constrain the form of the rules of $\mathcal{R}$ to forbid the construction of unbounded partial applications during rewriting. In order to constructively define such $\mathcal{K}$ (then $\mathcal{K}^n$), we take advantage of a feature of most functional programming languages: a *type system*.

4.1 Types

Our definition of types is inspired of [1]. Let $\mathscr{B}$ be a non-empty set of *base types*. The set of *simple types* $\mathscr{T}$ (or just *types*) is inductively defined as the least set containing $\mathscr{B}$ and all function types, *i.e.* $A \to B \in \mathscr{T} \Leftarrow A, B \in \mathscr{T}$. The function type constructor $\to$ is assumed to be right-associative. The *arity* of a type A is inductively defined on the structure of A by:

$$\begin{aligned} ar(A) &= 0 && \Leftarrow A \in \mathscr{B} \\ ar(A \to B) &= 1 + ar(B) && \Leftarrow A \to B \in \mathscr{T} \end{aligned}$$

Instead of using alphabets, in a typed terms environment we use *signatures* $\mathcal{F} = \mathcal{C} \cup \{@\}$ where $\mathcal{C}$ is a set of *constructor* symbols associated to a unique type and @ the application symbol (with no type). We also assign a type to every

variables. We note $f : A$ the symbol f of type A and $t : A$ a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ of type A. The set of all *well typed terms* $\mathcal{W}(\mathcal{F}, \mathcal{X})$ is inductively defined by:

$$\frac{f : A \in \mathcal{F}^0}{f : A \in \mathcal{W}(\mathcal{F}, \mathcal{X})} \qquad \frac{x : A \in \mathcal{X}}{x : A \in \mathcal{W}(\mathcal{F}, \mathcal{X})}$$

$$\frac{t_1 : A \to B \in \mathcal{W}(\mathcal{F}, \mathcal{X}) \qquad t_2 : A \in \mathcal{W}(\mathcal{F}, \mathcal{X})}{@(t_1, t_2) : B \in \mathcal{W}(\mathcal{F}, \mathcal{X})}$$

Definition 4 (Typed Rewriting). *A Typed TRS is a set of rules of the form*

$$l : A \to r : A$$

where $l : A$ and $r : A$ are two well typed terms of $\mathcal{W}(\mathcal{F}, \mathcal{X})$. A Typed TRS $\mathcal{R}$ derives the rewriting relation $\to_{\mathcal{R}}$ such that for all typed terms $s : B$ and $t : B$ of $\mathcal{W}(\mathcal{F}, \mathcal{X})$, $s \to_{\mathcal{R}} t$ iff there exists a rule $l : A \to r : A$ of $\mathcal{R}$, a position $p \in Pos(s)$ and a substitution $\sigma \in \mathcal{X} \to \mathcal{W}(\mathcal{F}, \mathcal{X})$ such that: $l\sigma : A = s|_p : A$ and $t : B = s[r\sigma]_p : B$.

In the same way, an equation $s = t$ is well typed if both s and t have the same type. In the rest of this paper we only consider well typed equations and TRSs.

Types can be useful only if they tells us something about how a term can be rewritten. For instance we expect the term $@(f : A \to B, x : A) : B$ to be rewritten by every *complete* (no partial function) TRS $\mathcal{R}$ if $ar(A \to B) = 1$. This is the conventional behavior of functional programs, but we need to be sure that the rewriting system we consider respects this behavior.

Definition 5 (Functional TRS). *A Functional TRS, with Higher-Order, is composed of rules of the form $l : B \to r : B$ where r is a well typed term, and l a term of F where F is inductively defined by:*

$$\begin{aligned} f(t_1 : B_1, \ldots, t_n : B_n) : B \in F \Leftarrow f : B_1 \to \cdots \to B_n \to B \in \mathcal{F}^n \\ t_1 : B_1, \ldots, t_n : B_n \in \mathcal{W}(\mathcal{C}, \mathcal{X}) \\ @(t : A \to B, x : A) : B \in F \Leftarrow t \in F, x \in \mathcal{W}(\mathcal{C}, \mathcal{X}) \end{aligned}$$

A functional TRS is complete *if for all term $t = @(t_1, t_2) : A$ such that $ar(A) = 0$, it is possible to rewrite t using $\mathcal{R}$. In other words, there are no partial functions.*

Combining typed terms and functional TRSs gives us some useful informations. For a given signature $\mathcal{F}$, in addition of type arity, the *order* of a type A noted $ord(A)$ is inductively defined on the structure of A by:

$$\begin{aligned} ord(A) &= max\{ord(f) \mid f : \cdots \to A \in \mathcal{C}^n\} \\ ord(A \to B) &= max\{ord(A) + 1, ord(B)\} \end{aligned}$$

where $ord(f : A_1 \to \ldots \to A_n \to A) = max\{ord(A_1), \ldots, ord(A_n)\}$ with $ord(A) = 0$. For instance $ord(int) = 0$, $ord(int \to int) = 1$. If we have

$cons : int \to list \to list \in \mathcal{C}$ then $ord(list) = 0$. However if $cons : (int \to int) \to list \to list \in \mathcal{C}$ then $ord(list) = 1$. In the first case the type $list$ designate lists of int, in the second case lists of $int \to int$ which makes a difference for us. In the first case a term of type $list$ once reduced cannot contain any symbol @ where in the second case it can.

Lemma 3 (The order function). *Let $\mathcal{R}$ be a weakly terminating functional TRS and A a type such that $ord(A) = 0$. Then for all terms t of type A it is rewritten into a term with no partial application:*

$$\forall s \in IRR(\mathcal{R}), \quad t \to_{\mathcal{R}}^{*} s \Rightarrow s \in \mathcal{B}^0(\mathcal{C})$$

Proof. By induction on the length of the rewriting path. If $t = s$ then we proceed by induction on the structure of s. Since $ord(A) = 0$, we can't have $s = @(f, u)$. Otherwise because $\mathcal{R}$ is a functional TRS we would have $s \notin IRR(\mathcal{R})$ which contradict the hypothesis. Thus $s = f$ with $f : B \in \mathcal{F}^0$ and by definition $s \in \mathcal{B}^0(\mathcal{C})$.

Now if $t \to_{\mathcal{R}}^{k+1} s$. If $t = @(f, u)$ then we know that there is $v : A$ such that $t \to_{\mathcal{R}} v$ since $\mathcal{R}$ is a functional TRS. Then $u \to_{\mathcal{R}}^{k} s$ and by hypothesis of induction, $s \in \mathcal{B}^0(\mathcal{C})$. If $t = f$, as previously we have by definition $s \in \mathcal{B}^0(\mathcal{C})$.

4.2 Terminating Programs

Recall that we want to define a set $\mathcal{K}^n \subseteq \mathcal{T}(\mathcal{F})$ and a class of TRS $\mathcal{K}$ such that (i) $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^n$ and (ii) $L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E}) \subseteq \mathcal{K}^n_{\trianglerighteq}$. Assuming that $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n_{\trianglerighteq}$ we then instantiate Theorem 1 with $\mathcal{L} = \mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R})$ and prove termination. A TRS is in our class if for all rule $l \to r \in \mathcal{R}$, r is in the set $\mathcal{K}$. By constraining the form of the right hand side of each rule of $\mathcal{R}$, $\mathcal{K}$ defines a class of TRS that cannot construct unbounded partial application during rewriting. The definition of $\mathcal{K}$ takes advantage of the defined type system and Lemma 3. $\mathcal{K}$ is inductively defined by:

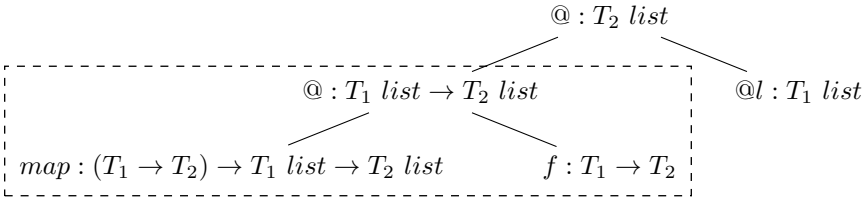
$$\underline{x} : A \in \mathcal{K} \Leftarrow \underline{x} : A \in \mathcal{X}$$
$$f(t_1, \ldots, t_n) : A \in \mathcal{K} \Leftarrow f \in \mathcal{C}^n \wedge t_1, \ldots, t_n \in \mathcal{K}$$
$$@(t_1 : A \to B, t_2 : A) : B \in \mathcal{K} \Leftarrow t_1 \in \mathcal{Z}, t_2 \in \mathcal{K} \wedge B \in \mathscr{B} \quad (1)$$
$$@(t_1 : A \to B, t_2 : A) : B \in \mathcal{K} \Leftarrow t_1, t_2 \in \mathcal{K} \wedge ord(A) = 0 \quad (2)$$

The last two rules ensure that for an application $@(t_1, t_2)$ we have either: (1) a total application, and the whole term can be rewritten; or (2) a partial application where t_2 can be rewritten into a term of $\mathcal{B}^0$ (Lemma 3). In the first case, $\mathcal{Z}$ is defined by:

$$t \in \mathcal{Z} \Leftarrow t \in \mathcal{K}$$
$$@(t_1, t_2) \in \mathcal{Z} \Leftarrow t_1 \in \mathcal{Z}, t_2 \in \mathcal{K}$$

Its purpose is to allow a partial applications inside the total application of a multi-parameter function.

Example 2. Let's consider the classical *map* function. A typical call to this function that may appear in $\mathcal{R}$ is $@(@(map, f), l)$, where f is a function, and l a list.



The whole term belongs to $\mathcal{K}$ but its subterm $@(map, f : A) : B$ belongs to $\mathcal{K}$ since $ar(B) \neq 0$ and $ord(A) \neq 0$. This subterm is a partial application, but there is no risk of stacking partial applications here since it is part of a complete call (to the *map* function).

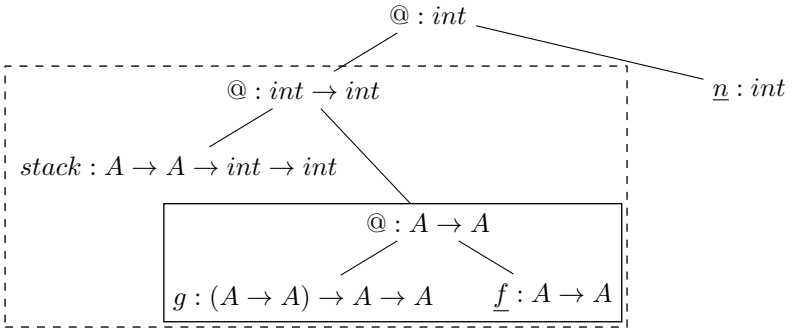
Example 3. Let's consider the function *stack* defined by:

$$@(@(stack, \underline{x}), 0) \to \underline{x}$$
$$@(@(stack, \underline{x}), S(\underline{n})) \to @(@(stack, @(g, \underline{x})), \underline{n})$$

Here g is a function of type $(A \to A) \to A \to A$. The *stack* function returns a stack of partial applications whose height is equal to the input parameter:

$$@(@(stack, f), \underbrace{S(S(S \dots S}_{k}(0) \dots))) \to^*_{\mathcal{R}} \underbrace{@(g, @(g, @(g, \dots @(}_{k}g, f) \dots)))$$

Since the depth of partial applications stacks in the output language is not bounded, with no equations on the @ symbol, the completion algorithm may not terminate. However note that the term $@(@(stack, @(g, \underline{x})), \underline{n})$ (in $\mathcal{R}$) is not in $\mathcal{K}$, so the TRS should no be accepted. It is a totally applied application. Hence $@(stack, @(g, \underline{x}))$ should be in $\mathcal{Z}$ and $\underline{n}$ in $\mathcal{K}$. Then by definition of $\mathcal{Z}$ we should have $@(g, \underline{x})$ in $\mathcal{K}$. However this time g is not applied totally and $ord(A \to A) > 0$ (where $A \to A$ is the type of $\underline{x}$).



We define $\mathcal{K}^n$ as $\{t\sigma \mid t \in \mathcal{K}, \quad \sigma : \mathcal{X} \mapsto \mathcal{B}^n \cap IRR(\mathcal{R})\}$ and claim that if for all rule $l \to r$ of the functional TRS $\mathcal{R}$, $r \in \mathcal{K}$ and if $L(\mathcal{A}) \subseteq \mathcal{K}^n$ then we can use Theorem 1 to prove that the completion of $\mathcal{A}$ with $\mathcal{R}$ terminates.

Theorem 2. *Let $\mathcal{A}$ be a $\mathcal{K}^n$-coherent REFD tree automaton, $\mathcal{R}$ a weakly-terminating functional TRS such that for all rule $l \to r \in \mathcal{R}, r \in \mathcal{K}$ and E a set of equations. Let $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. If $E = E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$ then the completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates.*

In this theorem, B is a fixed upper bound of the arity of all the types of the program. $\mathcal{K}^n$-coherence is defined later but basically says that $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n$. The proof of this theorem is done in this section in three steps:

- First we prove that $\mathcal{K}^n$ is closed by $\to_{\mathcal{R}}$.
- We prove that the completion, and in particular the application of the equations, also stays in $\mathcal{K}^n$ after each step, so that $\mathcal{L}_{\trianglerighteq}(\mathcal{A}^*)$ is in $\mathcal{K}^n$.
- Then we prove that there are a finite number of normal form of $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R})$ w.r.t $\vec{E^c_{\mathcal{L}}}$.
- Finally we use those three properties combined, and instanciate Theorem 1 with $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ to prove Theorem 2.

To prove that $\mathcal{K}^n$ is closed by $\to_{\mathcal{R}}$, we need some intermediate properties over $\mathcal{K}^n$. In particular make sure that, when we apply a rule, the considered substitution gives for each variable a term of $\mathcal{K}^n$: all rules are applied with a substitution of $\mathcal{K}^n$. First let us recall that in a functional TRS, each rule is of the form $l \to r$ with $l \in F$ (c.f. Definition 5) and $r \in \mathcal{K}$.

Lemma 4. *Let $\mathcal{R}$ be a functional TRS. For all well typed constructor terms $t \in \mathcal{W}(\mathcal{C}, \mathcal{X})$, For all terms $s \in \mathcal{K}^n$, if there exists σ such that $t\sigma = s$, then for all $x \in Var(t)$, $\sigma(x) \in \mathcal{K}^n$.*

Proof. By induction on t.

- $t = x$. If $t\sigma = s$, $s \in \mathcal{K}^n$, then $\sigma(x) = s$ and $\sigma(x) \in \mathcal{K}^n$.
- $t = f(t_1, \ldots, t_n), f \in \mathcal{C}^n$. If $t\sigma = s$, then $s = f(s_1, \ldots, s_n) = f(t_1\sigma, \ldots, t_n\sigma)$. By hypothesis of induction, for all t_i, for all $x \in Var(t_i), \sigma(x) \in \mathcal{K}^n$. Since $Var(t) = \bigcup_{i=1}^n Var(t_i)$, then for all $x \in Var(t), \sigma(x) \in \mathcal{K}^n$.

Lemma 5. *Let $\mathcal{R}$ be a functional TRS. For all terms $t \in F$, for all terms $s \in \mathcal{K}^n$, if there exists σ such that $t\sigma = s$, then for all $x \in Var(t)$, $\sigma(x) \in \mathcal{K}^n$.*

Proof. By induction on t.

- $t = f(t_1, \ldots, t_m), f \in \mathcal{F}^m \wedge t_1, \ldots, t_m \in \mathcal{W}(\mathcal{C}, \mathcal{X})$. For all t_i, using Lemma 4 we get for all variables $x \in Var(t_i), \sigma(x) \in \mathcal{K}^n$. Since $Var(t) = \bigcup_{i=1}^m Var(t_i)$, then for all $x \in Var(t), \sigma(x) \in \mathcal{K}^n$.
- $t = @(t_1, t_2)$ with $t_1 \in F$ and $t_2 \in \mathcal{W}(\mathcal{C}, \mathcal{W})$. Using Lemma 4 we get for all variables $x \in Var(t_2)$, $\sigma(x) \in \mathcal{K}^n$. By induction hypothesis, for all variables $x \in Var(t_1)$, $\sigma(x) \in \mathcal{K}^n$. Since $Var(t) = Var(t_1) \cup Var(t_2)$, then for all variables $x \in Var(t), \sigma(x) \in \mathcal{K}^n$.

Lemma 6. *Let $\mathcal{R}$ be a functional TRS. For all rules $l \to r \in \mathcal{R}$, For all terms $t \in \mathcal{K}^n$, if there exists σ such that $l\sigma = t$, then for all $x \in Var(l)$, $\sigma(x) \in \mathcal{K}^n$.*

In other words, each time a rule is used, we know that all variables are substituted with a term of $\mathcal{K}^n$.

Proof. Since $\mathcal{R}$ is a functional TRS, we have $l \in F$. Using Lemma 5 we have for all $x \in Var(l)$, $\sigma(x) \in \mathcal{K}^n$.

Lemma 7. *For all $t \in \mathcal{K}^n$, for all rules $l \to r \in \mathcal{R}$ if there exists a position p and a substitution σ with $l\sigma = t|_p$, then $t|_p \in \mathcal{K}^n$, and for all terms $s \in \mathcal{K}^n$, $t[s]_p \in \mathcal{K}^n$.*

Proof. By induction on t.

- $t \in \mathcal{B}^n(\mathcal{C}) \cap IRR(\mathcal{R})$. Since $t \in IRR(\mathcal{R})$, there is no p and σ s.t. $l\sigma = t|_p$.
- $t = f(t_1 : T_1, \ldots, t_n : T_n) : T, arity(T) = 0$. Two cases:
  - if $p = \lambda$, then $t|_p = t$, and $t \in \mathcal{K}^n$. $t[s]_p = s$, $s \in \mathcal{K}^n$.
  - if $p = i.q$, Since $t_i \in \mathcal{K}^n$, by induction hypothesis we have $t_i|_q \in \mathcal{K}^n$ and $t_i[s]_q \in \mathcal{K}^n$.
    $t|_p = t_i|_q$, so $t|_p \in \mathcal{K}^n$. $t[s]_p = f(t_1, \ldots, t_i[s]_q, \ldots, t_n)$, and since $t_i[s]_q \in \mathcal{K}^n$, $t[s]_p \in \mathcal{K}^n$.
- $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ with $order(T_1) = 0, \ldots, order(T_n) = 0$. We can use exactly the same reasoning (note that the order of the arity of each type never change).
- $t = @(t_1 : T_1, t_2 : T_2) : T$ with $order(T_2) = 0, t_1, t_2 \in \mathcal{K}$. We can use exactly the same reasoning.
- $t = @(t_1 : T_1, t_2 : T_2) : T$ with $arity(T) = 0, t_1 \in Z, t_2 \in \mathcal{K}$. We can use the same reasoning if $p = \lambda$ or $p = 2.q$. If $p = 1.q$ we have $t_1 \in \mathcal{Z}^n$ and $t|_p = t_1|_q$. Let's prove by induction on t_1 that we have $t_1|_q \in \mathcal{K}^n$, and $t_1[s]_q \in \mathcal{Z}^n$.
  - $t_1 \in \mathcal{K}^n$. By induction hypothesis (on t), $t_1|_q \in \mathcal{K}^n$, $t_1[s]_q \in \mathcal{K}^n$, and since $\mathcal{K}^n \subseteq \mathcal{Z}^n$, $t_1[s]_q \in \mathcal{Z}^n$.
  - $t_1 = @(t_1' , t_2') : T'$.
    - If $q = \lambda$, then since $\mathcal{R}$ is a functional TRS, $arity(T') = 0$, which means $t_1 \in \mathcal{K}^n$, and by induction hypothesis (on t), $t_1|_q \in \mathcal{K}^n$ and $t_1[s]_q \in \mathcal{K}^n$. So $t_1[s]_q \subseteq \mathcal{Z}^n$.
    - If $q = 1.q'$, then by induction hypothesis on t_1, $t_1|_q \in \mathcal{K}^n$ and $t_1[s]_q \in \mathcal{Z}^n$.
    - If $q = 2.q'$, then by induction hypothesis on t, $t_1|_q \in \mathcal{K}^n$ and $t_1[s]_q \in \mathcal{K}^n$. So $t_1[s]_q \subseteq \mathcal{Z}^n$.

  $t[s]_p = @(t_1[s]_q, t_2)$, and since $t_1[s]_q \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$ and $arity(T) = 0$, then by definition $t[s]_p : T \in \mathcal{K}^n$.

Lemma 8 ($\mathcal{K}^n$ is closed by $\to_\mathcal{R}$). *Let assume that for all rules $l \to r \in \mathcal{R}$ we have $r \in \mathcal{K}$. For all $t \in \mathcal{K}^n$, for all u such that $t \to_\mathcal{R} u$ we have $u \in \mathcal{K}^n$.*

Proof. By induction on t.

- If $t \in \mathcal{B}^n \cap IRR(\mathcal{R})$ then t is irreducible.

- If $t = f(t_1, \ldots, t_n), f \in \mathcal{F}^n$. For all rules $l \to r \in \mathcal{R}$ and position p such that $l\sigma = t|_p$ and $u = t[r\sigma]_p$. By definition of functional TRS, there is no rule for such term at the root, p cannot be λ. So $p = p'.q$. Let's define $t' = t|p'$, and $u' = t'[r\sigma]_q$. We have $u = t[u']_{p'}$. By induction hypothesis, $u' \in \mathcal{K}^n$, so using Lemma 7 we have $t[u']_{p'} \in \mathcal{K}^n$.
- $t = @(t_1 : A \to B, t_2 : A) : B$ with $order(A) = 0, t_1, t_2 \in \mathcal{K}^n$. We can use exactly the same reasoning. For all rules $l \to r \in \mathcal{R}$ and position p such that $l\sigma = t|_p$ and $u = t[r\sigma]_p$. Two cases:
  - if $p = \lambda$, for all $x \in Var(l)$, $t \rhd x\sigma$, which implies $x\sigma \in \mathcal{K}^n$ (Using Lemma 6). Finally, $r\sigma \in \mathcal{K}^n$.
  - if $p = p'.q$, we can use the same reasoning as above.
- $t = @(t_1, t_2) : B$ with $arity(B) = 0, t_1 \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$. For all rules $l \to r \in \mathcal{R}$ and position p such that $l\sigma = t|_p$ and $u = t[r\sigma]_p$. Two cases:
  - if $p = \lambda$, we can use the same reasoning as above.
  - if $p = 1.q$, using Lemma 7 again we have $t_1|_q \in \mathcal{K}^n$, and by induction hypothesis, $t'_1 = t_1[r\sigma]_q \in \mathcal{K}^n$, thus $@(t'_1, t_2) \in \mathcal{K}^n$.
  - if $p = 2.q$, let's define $t'_2 = t_2[r\sigma]_q$. We have $u = @(t_1, t'_2)$. By induction hypothesis, $t'_2 \in \mathcal{K}^n$, so $@(t_1, t'_2) \in \mathcal{K}^n$.

In the same way we can prove that $\mathcal{Z}^n$ is closed by $\to_{\mathcal{R}}$. To prove that after each step of completion, the recognized language stays in $\mathcal{K}^n$, we first need some assumptions over the considered automaton: we need it to be $\mathcal{K}^n$-coherent.

Definition 6 ($\mathcal{K}^n$-coherence). *Let $\mathcal{L} \subseteq \mathcal{W}(\mathcal{F})$ and $n \in \mathbb{N}$. $\mathcal{L}$ is $\mathcal{K}^n$-coherent if*

$$\mathcal{L} \subseteq \mathcal{K}^n \ \vee \ \mathcal{L} \subseteq \mathcal{Z}^n \setminus \mathcal{K}^n$$

By extension we say that a tree-automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ is $\mathcal{K}^n$-coherent if the language recognized by all states $q \in \mathcal{Q}$ is $\mathcal{K}^n$-coherent.

Our goal is to show that $\mathcal{K}^n$-coherence is preserved during completion, by $\mathcal{C}_{\mathcal{R}}(\cdot)$ (Lemma 13) then by $\mathcal{S}_E(\cdot)$ (Lemma 14).

Lemma 9. *For all context C and terms $s : T, t : T$ such that $s, t \in \mathcal{Z}^n \setminus \mathcal{K}^n$ or $s, t \in \mathcal{K}^n$, $C[s] \in \mathcal{K}^n \iff C[t] \in \mathcal{K}^n$.*

Proof. This can be done by induction on the context C by seeing in the definition of K that we can swap a subterm of a $\mathcal{K}^n$ term as long as the type of the subterm is the same, and that a $\mathcal{K}^n$ term is not replaced by a $\mathcal{Z}^n$ term.

Lemma 10 (Linking two states together preserves $\mathcal{K}^n$-coherence). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta' \rangle$ such that $\Delta' = \Delta \cup \{q_a \to q_b\}$ and $\mathcal{L}(\mathcal{A}, q_b) \subseteq \mathcal{K}^n \iff \mathcal{L}(\mathcal{A}, q_a) \subseteq \mathcal{K}^n$, with both languages sharing the same type T. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.*

Proof. For all state $q \in \mathcal{Q}$, for all term $s, t \in \mathcal{K}^n_{\rhd}$ such that we have $C[t] \to^*_{\Delta} C[q_a] \to_{\Delta'} C[q_b] \to^*_{\Delta} q$ with $C[s] \to^*_{\Delta} C[q_b] \to^*_{\Delta} q$. $C[t]$ is recognized in q with only one transition $q_a \to q_b$. Let's prove that $\mathcal{L}(\mathcal{A}, q) \in \mathcal{K}^n \iff C[t] \in \mathcal{K}^n$. If

$s \in \mathcal{K}^n$ (or $t \in \mathcal{K}^n$), then since $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $\mathcal{L}(\mathcal{A}, q_b) \subseteq \mathcal{K}^n$ (or $\mathcal{L}(\mathcal{A}, q_a) \subseteq \mathcal{K}^n$). Then since by hypothesis $\mathcal{L}(\mathcal{A}, q_b) \subseteq \mathcal{K}^n \iff \mathcal{L}(\mathcal{A}, q_a) \subseteq \mathcal{K}^n$, $s \in \mathcal{K}^n$ *and* $t \in \mathcal{K}^n$. Using Lemma 9 we have that $C[s] \in \mathcal{K}^n \iff C[t] \in \mathcal{K}^n$. If $\mathcal{L}(\mathcal{A}, q) \in \mathcal{K}^n$ then $C[s] \in \mathcal{K}^n$ and then $C[t] \in \mathcal{K}^n$. We can use the exact same reasoning if $s, t \in \mathcal{Z}^n \setminus \mathcal{K}^n$. We can generalize this reasoning to any number of $q_a \to q_b$ transitions in the derivation paths and see that $\mathcal{L}(\mathcal{A}', q)$ remains $\mathcal{K}^n$-coherent, for any state q. $\mathcal{A}'$ is $\mathcal{K}^n$ coherent.

To prove that the normalization preserves $\mathcal{K}^n$-coherence, we proceed by induction on the number of symbols of $\mathcal{F}$ in the considered term. For the sake of readability, this number is denoted $|t|$ in the proof, for all terms t (for instance $|f(g(q_1), q_2)| = 2$, for the two symbols f and g).

Lemma 11 (Normalization preserves $\mathcal{K}^n$-coherence). *For all $k \in \mathbb{N}, k > 0$, for all $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ such that $\Delta' = \Delta \cup Norm_\Delta(t \to q)$ with $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}), |t| = k$ and $Q' = Q \cup \tilde{Q} \cup \{q\}$ where $\tilde{Q}$ is the set of new states introduced by $Norm_\Delta(t \to q)$. If t is $\mathcal{K}^n$-coherent and $q \in \mathcal{Q} \Rightarrow t \to q \in \Delta$ then if $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.*

Proof. By induction on k.

- $k = 1$. Then $t = f(q_1, \ldots, q_m)$ for $f \in \mathcal{F}^m$ and $q_1, \ldots, q_m \in \mathcal{Q}$ and $Norm_\Delta(t \to q) = \{f(q_1, \ldots, q_m) \to q\}$. If $q \in \mathcal{Q}$ then $t \to q \in \Delta$, $\mathcal{A} = \mathcal{A}'$ so $\mathcal{A}'$ is $\mathcal{K}^n$-coherent. If $q \notin \mathcal{Q}$ then q is a new state so $\mathcal{L}(\mathcal{A}', q) = \mathcal{L}(\mathcal{A}, t)$ with t $\mathcal{K}^n$-coherent, thus $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.
- $k + 1$. Then there exists a context C such that $t = C[f(q_1, \ldots, q_m)]$ with $q_1, \ldots, q_m \in \mathcal{Q}$. So $Norm_\Delta(t \to q) = \{f(q_1, \ldots, q_m) \to q'\} \cup Norm_{\Delta \cup \{f(q_1, \ldots, q_m) \to q\}}(C[q'] \to q)$ where $f(q_1, \ldots, q_m) \to q' \in \Delta$ or q' is a new state if there is no $q'' \in \mathcal{Q}, f(q_1, \ldots q_n) \to q'' \in \Delta$.
  Note that $\{f(q_1, \ldots, q_m) \to q'\} = Norm_\Delta(f(q_1, \ldots, q_m) \to q')$, and according to Lemma **??**, we have $f(q_1, \ldots, q_m)$ $\mathcal{K}^n$-coherent. Let $\mathcal{A}'' = \langle \mathcal{F}, \mathcal{Q}'', \mathcal{Q}_f, \Delta'' \rangle$ where $\mathcal{Q}'' = \mathcal{Q} \sqcup \{q'\}$, and $\Delta'' = \Delta \cup \{Norm_\Delta(t \to q)\}$. $|f(q_1, \ldots, q_m)| = 1$. Using the same reasoning as for $k = 1$ we can deduce that $\mathcal{A}''$ is $\mathcal{K}^n$-coherent. $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta \rangle$ with $\mathcal{Q}' = \mathcal{Q}'' \cup \tilde{\mathcal{Q}} \cup \{q\}$ and $\Delta' = \Delta'' \cup Norm_{\Delta''}(t \to q)$. $|t| = k$. By hypothesis of induction, since $\mathcal{A}''$ is $\mathcal{K}^n$-coherent, $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.

Lemma 12 (Solving a critical-pair preserves $\mathcal{K}^n$-coherence). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ that contains the critical pair $\langle l \to r, q, \sigma \rangle$ and $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta \cup \Delta' \rangle$ where Δ' follows the definition **??** and $\mathcal{Q}' = \mathcal{Q} \cup \tilde{\mathcal{Q}}$. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.*

Proof. By definition of a critical pair we have $l\sigma \to_\Delta^* q$ and $l\sigma \to_\mathcal{R} r\sigma$. Since $\mathcal{K}^n$ is closed by $\to_\mathcal{R}$ we have $l\sigma \in \mathcal{K}^n \Rightarrow r\sigma \in \mathcal{K}^n$.

- If $\Delta' = \{q' \to q\}$ then there exists $q' \in Q$ s.t. $r\sigma \to_\Delta^* q'$. Since $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $l\sigma \in \mathcal{K}^n \iff \mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{K}^n$, and $\mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{K}^n \Rightarrow r\sigma \in \mathcal{K}^n \Rightarrow \mathcal{L}(\mathcal{A}, q') \subseteq \mathcal{K}^n$. We can do the opposite demonstration do deduce that

$\mathcal{L}(\mathcal{A}, q') \in \mathcal{K}^n \Rightarrow \mathcal{L}(\mathcal{A}, q) \in \mathcal{K}^n$ by first showing that $\mathcal{Z}^n \setminus \mathcal{K}^n$ is closed by rewriting. So we have $\mathcal{L}(\mathcal{A}, q) \iff \mathcal{L}(\mathcal{A}, q')$. Then using Lemma 10 we have $\mathcal{A}'$ $\mathcal{K}^n$-coherent.
- If $\Delta' = Norm_\Delta(r\sigma \to q') \cup \{q' \to q\}$, let $\mathcal{A}''$ the tree automaton $\langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta \cup \Delta'' \rangle$ where $\Delta'' = Norm_\Delta(r\sigma \to q')$. $\mathcal{L}(\mathcal{A}, r\sigma)$ is $\mathcal{K}^n$-coherent since $\mathcal{A}$ is $\mathcal{K}^n$-coherent and $q' \notin \mathcal{Q}$. Using Lemma 11 we have $\mathcal{A}''$ $\mathcal{K}^n$-coherent. Note that $\Delta' = \Delta'' \cup \{q' \to q\}$, so we can use the same reasoning as for the first case to prove that $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.

Lemma 13 ($\mathcal{C}_\mathcal{R}(\mathcal{A})$ preserves $\mathcal{K}^n$-coherence). *Let $\mathcal{A}$ be a REFD tree automaton. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{C}_\mathcal{R}(\mathcal{A})$ is $\mathcal{K}^n$-coherent.*

Proof. $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, Join^{CP(\mathcal{R},\mathcal{A})}(\Delta) \rangle$. Let us proceed by induction on the structure of $CP(\mathcal{R}, \mathcal{A})$.

- $CP(\mathcal{R}, \mathcal{A}) = \emptyset$. Then $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \mathcal{A}$, which is $\mathcal{K}^n$-coherent.
- $CP(\mathcal{R}, \mathcal{A}) = \{\langle l \to r, q, \sigma \rangle\} \cup S$. $Join^{CP(\mathcal{R},\mathcal{A})}(\Delta) = Join^S(\Delta \cup \Delta')$. Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup \Delta \rangle$. By Lemma **??**, $\mathcal{A}'$ is $\mathcal{K}^n$-coherent. $CP(\mathcal{R}, \mathcal{A}') = S$ and $Join^{CP(\mathcal{R},\mathcal{A}')}(\Delta \cup \Delta') = Join^S(\Delta \cup \Delta')$ so $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \mathcal{C}_\mathcal{R}(\mathcal{A}')$. Thus $\mathcal{C}_\mathcal{R}(\mathcal{A})$ is $\mathcal{K}^n$-coherent.

Lemma 14 ($\mathcal{S}_E(\mathcal{A})$ preserves $\mathcal{K}^n$-coherence). *Let $\mathcal{A}, \mathcal{A}'$ two REFD tree automata, $\mathcal{R}$ a functional TRS and E a set of equations such that $E = E^r \cup E_n^c \cup E_\mathcal{R}$. If $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}$ is $\mathcal{K}^n$-coherent then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent and $\mathcal{S}_E(\mathcal{A})$ is $\mathcal{K}^n$-coherent.*

Proof. Let's name q_a and q_b the two states merged from $\mathcal{A}$ to $\mathcal{A}'$. Let $\mathcal{A}''$ be the tree automaton defined from $\mathcal{A}$ by $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup \{q_a \to q_b, q_b \to q_a\} \rangle$. Note that for all states $q \in \mathcal{Q} \setminus \{q_a\}$, $L(\mathcal{A}', q) = L(\mathcal{A}'', q)$.

Let's show that $\mathcal{A}''$ is $\mathcal{K}^n$-coherent instead of $\mathcal{A}'$. First let's show that $L(\mathcal{A}, q_a) \subseteq \mathcal{K}^n \iff L(\mathcal{A}, q_b) \subseteq \mathcal{K}^n$. Since q_a and q_b are being merged, it means that there exists two terms s and t such that $s \to_\Delta^* q_a$, $t \to_\Delta^* q_b$ and $s =_E t$. Here we have three cases.

1. $s =_{E^r} t$, then $s = t$ and $s \in \mathcal{K}^n \iff t \in \mathcal{K}^n$.
2. $s =_{E_n^c}$, then $t = s|_p$ for some p meaning that there is an equation $u = u|_p$ and a substitution σ such that $u\sigma = s$ and $u|_p\sigma = t$. Recall that we restrain ourselves to well-typed equations where $u \in \mathcal{W}(\mathcal{C}, \mathcal{X})$. Then using the definition of $\mathcal{K}^n$ we have $s \in \mathcal{K}^n \iff s|_p \in \mathcal{K}^n$.
3. $s =_{E_\mathcal{R}} t$, then $s \to^\mathcal{R} t$ and using Lemma 8 (and its equivalent with $\mathcal{Z}^n$) we have $s \in \mathcal{K}^n \iff t \in \mathcal{K}^n$.

So we have $s \in \mathcal{K}^n \iff t \in \mathcal{K}^n$. Now since $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $L(\mathcal{A}, q_a) \subseteq \mathcal{K}^n \iff L(\mathcal{A}, q_b) \subseteq \mathcal{K}^n$. By Lemma 10 we have $\mathcal{A}''$ $\mathcal{K}^n$-coherent. By extension since for all states $q \in \mathcal{Q} \setminus \{q_a\}$, $L(\mathcal{A}', q) = L(\mathcal{A}'', q)$, $\mathcal{A}'$ is $\mathcal{K}^n$-coherent and $\mathcal{S}_E(\mathcal{A})$ is $\mathcal{K}^n$-coherent.

By using Lemma 13 and Lemma 14, we can prove that the completion algorithm, which is a composition of $C_\mathcal{R}(\mathcal{A})$ and $\mathcal{S}_E(\mathcal{A})$, preserves $\mathcal{K}^n$-coherence.

Lemma 15 (Completion preserves $\mathcal{K}^n$-coherence). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ a functional TRS and E a set of equations. If $E = E^r \cup E_n^c \cup E_{\mathcal{R}}$ and $\mathcal{A}$ is $\mathcal{K}^n$-coherent then for all $k \in \mathbb{N}$, $\mathcal{A}^k_{\mathcal{R},E}$ is $\mathcal{K}^n$-coherent. In particular, $\mathcal{A}^*$ is $\mathcal{K}^n$-coherent.*

By construction we can prove that the depth of irreducible $\mathcal{K}^n_{\trianglerighteq}$ terms is bounded, which correspond to the following Lemma:

Lemma 16. *For all $t : T \in \mathcal{K}^n_{\trianglerighteq}$, $t : T \in IRR(\mathcal{R}) \Rightarrow t : T \in \mathcal{B}^{n+2B-arity(T)}$.*

Proof. By induction on $t : T$.

- $t : T \in \mathcal{K}^n$, then by Lemma **??** we have $t : T \in \overrightarrow{HO}^{n+B-arity(T)}_{\mathcal{R}}(\mathcal{C})$, thus $t : T \in \overrightarrow{HO}^{n+2B-arity(T)}_{\mathcal{R}}(\mathcal{C})$.
- $t = @(t_1 : T_1, t_2 : T_2) : T$ with $t_1 \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$. By Lemma **??** we have $t_2 : T_2 \in \overrightarrow{HO}^{n+B}_{\mathcal{R}}(\mathcal{C})$. By induction hypothesis we have $t_1 : T_1 \in \overrightarrow{HO}^{n+2B-arity(T_2)}_{\mathcal{R}}(\mathcal{C})$. Since $arity(T_2) = arity(T)+1$, $t_1 : T_1 \in \overrightarrow{HO}^{n+2B-arity(T)-1}_{\mathcal{R}}(\mathcal{C})$. No that know the depth of t_1 and t_2, we can deduce the depth of t by taking the maximum depth of t_1 and t_2, which gives us $t : T \in \overrightarrow{HO}^{max(n+2B-arity(T),n+B+1)}_{\mathcal{R}}(\mathcal{C})$. However $arity(T_2) \leq B$ implies $arity(T) < B$ and then $B - arity(T) - 1 \geq 0$. Thus $n + B + 1 \leq n + 2B - arity(T)$. Finally $t : T \in \overrightarrow{HO}^{n+2B-arity(T)}_{\mathcal{R}}(\mathcal{C})$.

4.3 Final proof

Now we can prove the final theorem.

Proof (Theorem 2). According to Lemma 15, for all $k \in \mathbb{N}$, $\mathcal{A}^k_{\mathcal{R},E}$ is $\mathcal{K}^n$-coherent. By definition this implies that $\mathcal{L}_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E}) \subseteq \mathcal{K}^n_{\trianglerighteq}$. Moreover, we know that $IRR(\mathcal{R}) \cap \mathcal{K}^n_{\trianglerighteq} \subseteq \mathcal{B}^{n+2B}$ and $\mathcal{R}$ is weakly-terminating, so for all term $s \in \mathcal{L}_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ there exists $t \in \mathcal{B}^{n+2B}$ such that $s \to^*_{\mathcal{R}} t$. Finally, since thee number of normal form of $\mathcal{B}^{n+2B}$ if finite w.r.t $\vec{E}$, according to Theorem 1 this implies that the completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates.

5 Equations Generation

The Theorem 2 requires many hypothesis that must be ensured before using the completion algorithm:

- $\mathcal{A}$ $\mathcal{K}^n$-coherent and REFD. These properties come naturally by building well-typed automata in the absence of polymorphism.
- $\mathcal{R}$ weakly-terminating.
- All left-hand sides rules of $\mathcal{R}$ is in $\mathcal{K}$. This can be checked statically by comparing the structure of each rule against the definition of $\mathcal{K}$. If it is not verified, we reject the program before starting the completion.

– $E = E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$. Since we want an automated verification method, E must be build automatically, not given by the user. The definitions of E^r and $E_{\mathcal{R}}$ are fixed by $\mathcal{R}$. On the opposite, there is no unique suitable set of contracting equations $E^c_{\mathcal{L}}$. Furthermore, if it is not chosen wisely, it could severely damage the precision of the completion. In most cases, it would become impossible to prove the correctness of the program.

In this section, we describe a method to generate all possible $E^c_{\mathcal{L}}$. Then we describe a simple verification algorithm able enumerating these $E^c_{\mathcal{L}}$ to check the correctness of a program.

5.1 Generating the contracting equations

For the sake of simplicity, we only present the case where $IRR(\mathcal{R}) \subseteq \mathcal{W}(\mathcal{C})$ (first-order results, i.e. no functions in the results). The problem is to find contracting equations for $\mathcal{W}(\mathcal{C})$ instead of $\mathcal{B}^{n+2B}$. This section defines $\mathbb{E}^k_c$ the set of all possible contracting equation sets for $\mathcal{W}(\mathcal{C})$, such that for each equation $u = u|_p$, $|u| \leq k+1$ where $|u|$ is the height of u. Our strategy to generate one set is to (i) generate every left-hand side of equation so that for each term $t \in \mathcal{W}(\mathcal{C})$ there exists a left-hand side u and a substitution σ such that $t = u\sigma$. (ii) find every equation of the form $u = u|_p$ where both sides have the same type.

To find every left-hand side we use the notion of *covering set of terms*, inspired by []. A set of terms C is covering for $t \in \mathcal{W}(\mathcal{C}, \mathcal{X})$ if for all substitution σ, there exists a term $s \in C$ and a substitution σ' such that $t\sigma = s\sigma'$. We define a function $CT_k(t)$ computing all the possible covering sets for $t \in \mathcal{W}(\mathcal{C}, \mathcal{X})$ where for all covering set $S \in CT_k(t)$, for all term $u \in S$, $|u| \leq |t| + k$.

$$
\begin{aligned}
CT_0(t) &= \{\{t\}\} \\
CT_{k+1}(x : A) &= \{\{x\}\} \cup \prod_{f : A_1 \to \cdots \to A_n \to A \in \mathcal{C}^n} CT_{k+1}(f(x_1, \ldots, x_n)) \\
CT_{k+1}(f(x_1 : A_1, \ldots, x_n : A_n)) &= \bigcup_{(t_1,\ldots,t_n) \in \Pi} \{f(t_1, \ldots, t_n)\} \\
&\text{where } \Pi = \prod_{1 \leq i \leq n} CT_k(x_i : A_i)
\end{aligned}
$$

From CT_k we can compute the set $\mathbb{E}^k_c$ of all the possible sets of contracting equations, where for each equation $u = u|_p$, $|u| \leq k+1$.

$$
\begin{aligned}
\mathbb{E}^k_c &= \prod_{A \in \mathcal{T}} \mathbb{E}^k_c(A) \\
\mathbb{E}^k_c(A) &= \{E(U) \mid U \in CT_k(x : A)\} \\
E(U) &= \{u = u|_p \mid p \neq \lambda \wedge u : A \in U \wedge u|_p : A\}
\end{aligned}
$$

where $\mathcal{T}$ is the set of all used types (types A such that there exists a constructor $f : \cdots \to A \in \mathcal{C}$). $\mathbb{E}^k_c(A)$ computes the different sets of contracting equations

for a single type by computing the possible covering sets U, and using $E(U)$ to convert it into an equation set.

5.2 Verifying a program

We define a verification function using the completion algorithm that checks a property φ on a given program. The program is represented by a TRS $\mathcal{R}$ respecting the hypothesis of Theorem 2. The initial call to the program with the inputs is represented by a tree automaton $\mathcal{A}$, respecting the hypothesis of Theorem 2. We define $\mathcal{L}_\varphi$ as the smallest language such that $\mathcal{A} \vdash \varphi \Rightarrow L(\mathcal{A}) \subseteq \mathcal{L}_\varphi$. In practice $\mathcal{L}_\varphi$ will be given by the user instead of the property itself. The verification algorithm is defined as follow:

Algorithm 1: Verification algorithm

Input : $\mathcal{R}$, $\mathcal{A}$, φ
Output: *Success* if $\mathcal{A}$ is correct w.r.t φ,
Fail if it is not correct w.r.t φ,
Maybe if it is not a regular property.

forall $k \in \mathbb{N}$ **do**
  $\mathcal{A}_k^* \leftarrow Completion_{\mathcal{R}}(\mathcal{A}_k)$;
  if $L(\mathcal{A}_k^*) \not\subseteq \mathcal{L}_\varphi$ **then**
    return Fail;
  else
    forall $E_c \in \mathbb{E}_c^k$ **do**
      $E \leftarrow E^r \cup E_{\mathcal{R}} \cup E_c$;
      $\mathcal{A}_k^* \leftarrow Completion_{\mathcal{R},E}(\mathcal{A}_k)$;
      if $L(\mathcal{A}_k^*) \subseteq \mathcal{L}_\varphi$ **then**
        $\mathcal{A}^* \leftarrow Completion_{\mathcal{R},E}(\mathcal{A})$;
        if $L(\mathcal{A}^*) \subseteq \mathcal{L}_\varphi$ **then**
          return Success;
    if *there is no* $E_c \in \mathbb{E}_c^k$ *such that* $\varphi \cap L(\mathcal{A}_k^*) = \emptyset$ **then**
      return Maybe;

This algorithm tries to find a set E_c for which the completion raise no counter example for φ ($L(\mathcal{A}^*) \subseteq \mathcal{L}_\varphi$). At each iteration, it tries to complete $\mathcal{A}^k$ the tree automaton recognizing every term of maximum height k recognized by $\mathcal{A}$. Since the language recognized by this automaton is finite and $\mathcal{R}$ is terminating, the set of reachable terms is finite and completion terminates. Furthermore, on finite initial languages completion computes exactly $\mathcal{R}^*(L(\mathcal{A}))$ [10]. If a counter example is found, then the verification failed. Otherwise it starts searching for a suitable equations set E_c. We know that such E_c should not introduce counter examples in the completion of $\mathcal{A}^k$. We also know that any equation $u = u|_p$ where $|u| > k$ cannot be applied during the completion of $\mathcal{A}^k$. The idea is then to find E_c in $\mathbb{E}_c^k$ that does not introduce a counter example in $\mathcal{A}^k$. Such E_c is a good candidate for the full completion of $\mathcal{A}$. If E_c does not introduce any counter example during the completion of $\mathcal{A}^k$ and $\mathcal{A}$, then the completion is a success and the program is verified. If there is no E_c introducing no counter example

in $\mathcal{A}^k$, then we cannot conclude (for instance if φ is not a regular property). If every candidate E_c introduce a counter example in the completion of $\mathcal{A}$, it starts all over again with a greater k.

If there is a counter example, this algorithm will find it for some k by completing $\mathcal{A}^k$ without equations. If the program is correct and there exists some E_c able to verify it, this algorithm will find it. Otherwise it returns *Maybe*. In every case it terminates, however it is a bruteforce algorithm which may takes some time before finding E_c. It does not scale if the minimal E_c contains too many equations.

6 Experiments

The verification technique described above has been integrated in the Timbuk library [9]. We present some verification experiments using this technique on several classical higher-order functions: *map*, $filter$, *exists* as well as higher-order sorting functions parameterized by an ordering function. We implemented a very naive version of equation generation where all possible equation sets E_c are enumerated. We describe the results that we have obtain on the introductory example. Recall that we wanted to prove that it is impossible to rewrite any term of the language $@(@(\mathit{exists}, \mathit{even}), @(@(\mathit{filter}, \mathit{odd}), l))$ (where l is any list of natural number) to *true*. This is described by the following Timbuk specification file, where `R1` is the TRS, `A0` is the initial language of terms, `TC` for equation generation. The section **Patterns** gives the term that should not be reachable, i.e. the term `true`.

```
Ops app:2 filter:0 o:0 s:1 nz:0 nil:0 cons:2 ite:0 true:0 false:0
    exists:0 even:0 odd:0
Vars F X Y Z U Xs
TRS R1
    app(app(app(ite,true),X),Y) -> X
    app(app(app(ite,false),X),Y) -> Y
    app(even,o) -> true
    app(odd,o) -> false
    app(even,s(X)) -> app(odd,X)
    app(odd,s(X)) -> app(even,X)
    app(app(filter,X),nil) -> nil
    app(app(filter,X),cons(Y,Z)) ->
        app(app(app(ite,app(X,Y)),cons(Y,app(app(filter,X),Z))),
              app(app(filter,X),Z))

    app(app(exists,X),nil) -> false
    app(app(exists,X),cons(Y,Z)) ->
        app(app(app(ite,app(X,Y)),true),app(app(exists,X),Z))

Automaton A0 States q0 q1 q2 q3 q4 q5 q6 q7 q8 q9 Final States q0
Transitions
    app(q5,q6)->q1   app(q1,q2)->q0    s(q9)->q9   o->q9
```

```
exists->q5      cons(q9,q4)->q4   nil->q4     app(q7,q8)->q3
filter->q7      app(q3,q4)->q2    odd->q8     even->q6
```

Automaton `TC` **States** `qb qn ql` **Final States** `qb qn ql` **Transitions**
`true->qb  false->qb  o->qn  s(qn)->qn  nil->ql  cons(qn,ql)->ql`

Patterns `true`

With this specification, Timbuk succeeds in proving that `true` is not reachable. It first generates a set of contracting equations E_c, then complete `A0` w.r.t. with `R1` and finally check that `true` is not recognized by the completed automaton. Here, the generated set E_c is $\{cons(F, cons(o, X)) = cons(o, X), cons(s(o), F) = F, s(s(F)) = F\}$ where F and Y are variables. A dozen of experiments can be found on Timbuk's web site[1]. Those experiments include several examples where counter-examples are automatically found and where equation generation fails. This is the case for the *map* function on trees. For this example, we know that there exists a set of contracting equations proving the property but a naive enumeration of equations is too costly to find it in a reasonable time. This algorithm should be improved to scale up in practice.

7 Related Work

For the past years, research on automated verification has been focused on first-order imperative programs, with the success of abstract interpretation and lots of existing tools such as ASTREE [3], SLAM [2], etc. There have been some attempts to use abstract interpretation in the case of higher-order functional languages [11], but the approach was to first lift down the program to first-order.

Liquid Types [17] and later Bounded Rafinement Types [22,21], and in the same time [6,5], is an attempt take advantage of the type system of functional languages and prove interesting properties on higher-order programs. Considering that type-checking is already a form of verification for very simple properties (e.g. "this expression has type A"), the idea is to extend the type specification language to express stronger properties. This has been successfully implemented in Haskell [23,20] and used to prove non trivial properties on higher-order programs. However the user still has to express the property he wants to prove using the type system, which can sometimes be tedious. In some cases, the user even has to specify what is apparent to intermediate lemmas to help the type checker. We are looking for a fully automated technique.

Model checking with Pattern Matching Recursion Schemes [15,19] (PMRS) is an other promising approach. Initially, PMRS were developed as an extension of Higher Order Recursion Schemes [16,13] to handle pattern matching and data structure commonly used in higher-order functional programs. Since its implementation in MoChi [14], the technique has evolved and now includes predicate

[1] http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/

abstraction (allowing verification of non-regular properties) and even modularity [18]. Despite its undeniable strengths, as far as we know, it is unable to verify programs such as Example 1. *** TG: Ca j'en suis sûr pour [15] mais pour le dire comme ça il faudrait être bien sûr que c'est aussi le cas pour [14,19,18]. On doit aussi pouvoir citer une réf sur un rapport de Yann qui montre pourquoi [15] ne peut pas traiter cet exemple *** Their technique is also capable to analyze non terminating functions. Dealing with non terminating functions can easily be done in our framework by loosing some precision. By adding contracting equations on function calls, it is possible to finitely approximate a diverging recursive function. Having such equation while preserving precision is an interesting research topic that we leave for future work.

Our technique is also close to [12] that uses regular over-approximations of the image of functions. However, the precision of their analysis is fixed though ours can automatically be adapted to the verification objective. In particular, their fixed precision is incompatible with all the examples of our experiments.

The tree automaton completion technique has been developed to be a fully automated technique able to verify such programs. Until now, the completion algorithm was guaranteed to terminate only in the case of first-order programs [10].

Building non regular over-approximations of reachable terms for TRS is also possible [4]. Though, the principle of their technique is close to tree automata completion, it is more difficult to tune the precision of the approximation. In particular, up to now, it cannot be automatically adapted to a given verification goal. Nevertheless, dealing with non regular over-approximations would permit, in theory, to deal with relational proof goals like the fact that the length of $@(@(\mathit{filter}, \mathit{odd}), l))$ is smaller or equal to the length of l. Thus, parameterized non regular approximations would be a very powerful verification tool.

8 Future Work

Smart generation of equations (voir fin partie experiments).

Polymorphism.

LTA: (V. Murat)... add abstract values in structures.

Theorem for free [24].

Implementation for OCaml.

Evaluation Strategies (Y. Salmon)

9 Conclusion

Conclusion.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)

2. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. (2002) 1–3
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003. (2003) 196–207
4. Boichut, Y., Chabin, J., Réty, P.: Towards more precise rewriting approximations. In: LATA'15. Volume 8977 of LNCS., Springer (2015) 652–663
5. Castagna, G., Nguyen, K., Xu, Z., Abate, P.: Polymorphic functions with set-theoretic types: part 2: local type inference and type reconstruction. In: POPL'15, ACM (2015)
6. Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., Padovani, L.: Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In: POPL'14, ACM (2014)
7. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: Tree automata techniques and applications. `http://tata.gforge.inria.fr` (2008)
8. Coq: The coq proof assistant reference manual: Version 8.6. (2016)
9. Genet, T., Boichut, Y., Boyer, B., Murat, V., Salmon, Y.: Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1 http://people.irisa.fr/Thomas.Genet/timbuk/.
10. Genet, T.: Termination criteria for tree automata completion. Journal of Logical and Algebraic Methods in Programming **85**(1) (2016) 3–33
11. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: verifying functional programs using abstract interpreters. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. (2011) 470–485
12. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theoretical Computer Science **375**(1-3) (2007) 120–136
13. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. (2009) 416–428
14. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. (2011) 222–233
15. Ong, C.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. (2011) 587–598
16. Ong, C.H.: On model-checking trees generated by higher-order recursion schemes. In: 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06), IEEE (2006) 81–90
17. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. (2008) 159–169

18. Sato, R., Kobayashi, N.: Modular verification of higher-order functional programs. In: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. (2017) 831–854
19. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. (2013) 75–86
20. Vazou, N.: Liquid Haskell: Haskell as a Theorem Prover. PhD thesis, University of California, San Diego, USA (2016)
21. Vazou, N., Bakst, A., Jhala, R.: Bounded refinement types. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. (2015) 48–61
22. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. (2013) 209–228
23. Vazou, N., Seidel, E.L., Jhala, R.: Liquidhaskell: experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014. (2014) 39–51
24. Wadler, P.: Theorems for free! In: FPCA. (1989)