Verifying Higher-Order Functions with Tree Automata

Thomas Genet, Timothée Haudebourg, Thomas Jensen

Verifying Higher-Order Functions 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 safety properties of higher-order functional programs. Tree automata are used to represent sets of reachable states and functional programs are modeled using term rewriting systems. From a tree automaton representing the initial state, a completion algorithm iteratively computes an automaton which over-approximates the output set of the program to verify. We identify a subclass of higher-order functional programs for which the completion is guaranteed to terminate. Precision and termination are obtained conjointly by a careful choice of equations between terms. The verification objective can be used to generate sets of equations automatically. Our experiments show that tree automata are sufficiently expressive to prove intricate safety properties and sufficiently simple for the verification result to be certified in Coq. These results extend state-of-the-art model-checking approaches for higher-order functional programs.

1 Introduction

Higher-order functions are an integral feature of modern programming languages such as Java, Scala or JavaScript, not to mention Haskell and Caml. Higher-order functions are useful for program structuring but pose a challenge when it comes to 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 [13] or Isabelle/HOL [28], or write a specification in a formalism such as Liquid Types [29] or Bounded Refinement Types [33,32] 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. The flow analysis of higher-order functions was studied by Jones [19] who proposed to model higher-order functions as term rewriting systems and use regular grammars to approximate the result. More recently, the breakthrough results of Ong *et al.* [27,21] and Kobayashi [22,24] show that combining abstraction with model checking techniques can be used with success to analyse higher-order functions automatically. Their approach 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 goals of higher-order functional verification using an approach based on the original term rewriting models of Jones. We present a formal verification technique based on Tree Automata Completion (TAC) [17], capable of checking a class of properties, called *regular properties*, of higher-order programs 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 the computed automaton. This completion-based approach also permits to *certify* automatically $\mathcal{A}^*$ in Coq [6], i.e. given $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{A}^*$, obtain the formal proof that $\mathcal{A}^*$ recognizes all terms reachable from $\mathcal{A}$ by $\mathcal{R}$-rewriting.

Example 1. The following term rewriting system $\mathcal{R}$ defines the $filter$ function along with the two predicates *even* and *odd* on Peano's natural numbers.

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

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

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

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

This function returns the input list where all elements not satisfying the input boolean function p are filtered out. Variables are underlined and the special symbol @ denotes function application where $@(f, x)$ means "x applied to f".

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 recognising 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 $\mathcal{A}^*$ recognising every term reachable from $L(\mathcal{A})$ (the set of terms recognised by 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. $true \notin L(\mathcal{A}^*)$. We denote by *regular properties* the family of properties provable on a regular set (or regular over-approximation) of reachable terms. In particular, regular properties do not count symbols in terms, nor relate subterm heights (for instance, a property comparing the length of the list before and after $filter$ is not regular)

Termination of the tree automata completion algorithm is not ensured in general [17]. For instance, if $\mathcal{R}^*(L(\mathcal{A}))$ is not regular, it cannot be represented as a tree automaton. In this case, the user can provide a set of *equations* that will force termination for first-order programs by introducing an approximation based on *equational abstraction* [25]: $\mathcal{R}^*(L(\mathcal{A})) \subseteq L(\mathcal{A}^*)$. 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 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 shortcomings with the following contributions:

- 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 class 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 paper is organised as follows: We describe the completion algorithm and how to use equations to ensure termination in Section 2. The technical contributions as described above are developed in Sections 3 to 5. In Section 6, we present a series of experiments validating our verification technique, and discuss certification of results. We present related work in Section 7. Section 8 concludes the paper.

2 Background: Term rewriting and tree automata

Terms. 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 $\mathcal{X}$ induces a set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that:

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

A *language* is a set of terms. A term t is *linear* if the multiplicity of each variable in t is at most 1, and *closed* if it contains no variables. The set of 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 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 λ is the empty word and "." 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 write $\mathcal{L}_\trianglerighteq$ for the language $\mathcal{L}$ and all its subterms. A *substitution* σ is an application of $\mathcal{X} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$, mapping variables to terms. 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 $\sigma = \{\Box \mapsto t\}$.

Term rewriting systems [1] provide a flexible way of defining functional programs and their 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

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

The term $cons(s(0), nil)$ is *irreducible* (no rule applies to it) and hence the result of the function call. We write $IRR(\mathcal{R})$ for the set of irreducible terms of $\mathcal{R}$.

Tree automata [12] 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, $\mathcal{Q}$ a finite set of states, $\mathcal{Q}_f$ the set of *final states*, and Δ 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 *recognised* by a state $q \in \mathcal{Q}$ if $t \to^*_\Delta q$. We write $L(\mathcal{A}, q)$ for the language of all terms recognised by q. A term t is recognised by $\mathcal{A}$ if there exists a final state $q \in \mathcal{Q}_f$ such that $t \in L(\mathcal{A}, q)$. In that case we write $t \in L(\mathcal{A})$. *E.g.*, the tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ with $\mathcal{F} = \{nil : 0, cons : 2, 0 : 0, s : 2\}$, $\mathcal{Q}_f = \{q_{list}\}$ and $\Delta =$

$$0 \to q_\mathbb{N} \qquad nil \to q_{list}$$
$$s(q_\mathbb{N}) \to q_\mathbb{N} \qquad cons(q_\mathbb{N}, q_{list}) \to q_{list}$$

recognises all lists of natural numbers.

An *ϵ-transition* is a transition of form $q \to q'$ where $q \in \mathcal{Q}$. A tree automaton $\mathcal{A}$ is *ϵ-free* if it contains no ϵ-transitions. We write $t \to^{\not\epsilon *}_\Delta q$ if t is recognised 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 $\mathcal{A}^*$ recognising 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 $\mathcal{R}^*(L(\mathcal{A}))$ 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* .

Definition 1 (Equation set). *In an equation set E, equations are of the form $l = r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. From E is derived the relation $=_E$ as the* smallest congruence *such that for all terms l, r and substitution σ we have:*

$$l = r \in E \quad \Rightarrow \quad l\sigma =_E r\sigma$$

The set of equivalence classes defined by $=_E$ on $\mathcal{T}(\mathcal{F})$ is noted $\mathcal{T}(\mathcal{F})/_{=E}$.

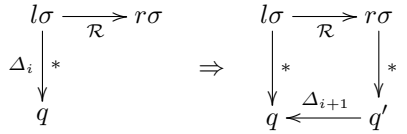
In this paper we also note $\vec{E}$ then TRS defined as $\{l \to r \mid l = r \in E\}$.

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 $\mathcal{A}^1_{\mathcal{R},E}, \mathcal{A}^2_{\mathcal{R},E}, \dots$ such that for all $i \in \mathbb{N}$:

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

It terminates when a fixpoint, written $\mathcal{A}^*$, 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 2. *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 $l\sigma$ recognised by $\mathcal{A}^i_{\mathcal{R},E}$ that can be rewritten by $\mathcal{R}$ into $r\sigma$. Completion adds the necessary states and transitions to $\mathcal{A}^{i+1}_{\mathcal{R},E}$ to obtain $r\sigma \in L(\mathcal{A}^{i+1}_{\mathcal{R},E})$.

Definition 3 (One step completion). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ be a left-linear TRS. The one step completed automaton is $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, Join^{CP(\mathcal{R},\mathcal{A})}(\Delta) \rangle$ where $Join^S(\Delta)$ is inductively defined by:*

$$Join^\emptyset(\Delta) = \Delta$$
$$Join^{\{\langle l \to r, q, \sigma \rangle\} \cup S}(\Delta) = Join^S(\Delta \cup \Delta')$$

where

$$\Delta' = \begin{cases} \{q' \to q\} & \text{if there exists } q' \in \mathcal{Q} \text{ s.t. } r\sigma \to^{\not e *}_\Delta q' \\ Norm_\Delta(r\sigma \to q') \cup \{q' \to q\} & \text{where } q' \text{ is a new state for } \Delta \text{ otherwise.} \end{cases}$$

The function $Norm_\Delta(t \to q)$ is used to *normalise* a transition. For instance in $s(0) \to q$, the left-hand side $s(0)$ is not a valid configuration. This transition is normalised into $\{0 \to q', s(q') \to q\}$ thanks to the normalisation function defined as follows.

Definition 4 (Normalisation). *Let Δ be a set of transitions defined on a set of states $\mathcal{Q}, t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q}$. Let $C[]$ be a non empty context of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})\mathcal{Q}$, $f \in \mathcal{F}$ of arity n, and $q, q', q_1, \ldots, q_n \in \mathcal{Q}$. The Normalisation function is inductively defined by:*

$$Norm_\Delta(f(q_1, \ldots, q_n) \to q) = \{f(q_1, \ldots, q_n) \to q\}$$
$$Norm_\Delta(C[f(q_1, \ldots, q_n)] \to q) = \{f(q_1, \ldots, q_n) \to q'\} \cup Norm_{\Delta \cup \{f(q_1,\ldots,q_n) \to q'\}}(C[q'] \to q)$$
$$\text{where} \begin{cases} f(q_1, \ldots q_n) \to q' \in \Delta \text{ or} \\ q' \text{ is a new state if there is no } q'' \in \mathcal{Q}, f(q_1, \ldots q_n) \to q'' \in \Delta \end{cases}$$

The simplification step merges all states q, q' for which there exist two terms $s =_E t$ and $s \to_\Delta^{\not\epsilon *} q$ and $t \to_\Delta^{\not\epsilon *} q'$. This results in an over-approximation of the original language: $L(\mathcal{A}) \subseteq L(\mathcal{S}_E(\mathcal{A}))$.

Definition 5 (Simplification relation). *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and E be a set of equations. For $s = t \in E, \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}), q_a, q_b \in \mathcal{Q}$ such that $s\sigma \to_\mathcal{A}^{\not\epsilon *} q_a, t\sigma \to_\mathcal{A}^{\not\epsilon *} q_b$ and $q_a \neq q_b$ then $\mathcal{A}$ can be simplified into $\mathcal{A}' = \mathcal{A}\{q_b \mapsto q_a\}$, denoted by $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$.*

Let $\mathcal{A}, \mathcal{A}'_1, \mathcal{A}'_2$ be tree automata and E be a set of equations. If $\mathcal{A} \rightsquigarrow_E^! \mathcal{A}'_1$ and $\mathcal{A} \rightsquigarrow_E^! \mathcal{A}_2$ then $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are equivalent modulo state renaming. $S_E(\mathcal{A})$ is the unique automaton (modulo renaming) $\mathcal{A}'$ such that $\mathcal{A} \rightsquigarrow_E^! \mathcal{A}'$.

Example 2. To illustrate how using an equation set over-approximates an automaton we use an example inspired by [**?**]. Let $E = \{cons(x, cons(y, nil)) = cons(y, nil)\}$ and $\mathcal{A}$ be the tree automaton with $Q_f = \{q_2\}$ and the set of transitions $\Delta = \{a \to q_0, nil \to q_1, cons(q_0, q_1) \to q_3, cons(q_0, q_3) \to q_2\}$. Hence $L(\mathcal{A}) = \{cons(a, cons(a, nil))\}$. Notice that we have the following:

$$\begin{array}{ccc} cons(q_0, cons(q_0, q_1)) & \overset{E}{=\!=} & cons(q_0, q_1) \\ \Delta \downarrow * & & \Delta \downarrow * \\ q_2 & & q_3 \end{array}$$

Thus, we can merge the states q_2 and q_3 into a single state q_2. The resulting set of transitions is $\Delta' = \{a \to q_0, nil \to q_1, cons(q_0, q_1) \to q_2, cons(q_0, q_2) \to q_2\}$. The resulting automaton $\mathcal{A}'$ recognizes any non-empty list so we have $L(\mathcal{A}') \supseteq L(\mathcal{A})$.

3 Termination of Tree Automata Completion

For a given TRS $\mathcal{R}$, initial state automaton $\mathcal{A}$ and set of equations E, the termination of the completion algorithm is undecidable in general, even with the use of equations. 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 (written **REFD** in the rest of the paper) for all k; (ii) every term of $\mathcal{A}^k_{\mathcal{R},E}$ can be rewritten into a term of $\mathcal{L}$ using $\mathcal{R}$ (for instance if $\mathcal{R}$ is terminating); (iii) $\mathcal{L}$ has a finite number of normal forms w.r.t $\vec{E}$. Completion is known to preserve $\not\epsilon$-reduceness and $\not\epsilon$-determinism if $E \supseteq E_r \cup E_\mathcal{R}$ [17] 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 that, in our verification setting, 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 6 (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.*

Example 3. Assume that $\mathcal{F} = \{s, 0\}$ where s has arity 1 and 0 has arity 0. The set $E^c_\mathcal{L} = \{s(x) = x\}$ is contracting for $\mathcal{L} = \mathcal{T}(\mathcal{F})$ because the set of normal of $\mathcal{T}(\mathcal{F})$ w.r.t.$\vec{E^c_\mathcal{L}} = \{s(x) \to x\}$ is the (finite) set $\{0\}$. The set $E^c_\mathcal{L} = \{s(s(x)) = x\}$ is contracting because normal forms of $\{s(s(x)) \to x\}$ are $\{0, s(0)\}$.

The contracting equations ensure that the completion algorithm will merge enough states during the simplification steps to terminate. Note that $E^c_{\mathcal{L}}$ cannot be empty, unless $\mathcal{L}$ is finite. First, we prove that it is possible to bound the number of states needed in $\mathcal{A}^*$ to recognise 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 $\mathcal{A}^*$ does not only recognises the output terms, additional states could have been added to the automaton in order to recognise intermediate computation terms. Theorem 1 states that with $E_{\mathcal{R}}$, the simplification steps will merge the states recognising the intermediate computation with the states recognising the outputs. If the latter set of states is finite then, using Lemma 2, $\mathcal{A}^*$ 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 [17]. Considering that an $\not\epsilon$-deterministic and $\not\epsilon$-reduced automaton that is ϵ-free is REFD by definition, Lemma 1 proves 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 only adds ϵ-transitions of form $q \to q'$ with q' a new state [17]. So, in $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$, for every transition $q \to q'$ such that $q \neq q'$ there exist 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 ϵ-transitions that remain after the simplification steps are of the form $q \to q$ and can be removed without altering the recognised language.

The following Lemma 2 states that with an REFD tree-automaton $\mathcal{A}$, the number of states needed in $\mathcal{S}_E(\mathcal{A})$ to recognise 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 G be 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 recognised 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 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 $t \to^*_{\mathcal{S}_E(\mathcal{A})} q'$, and since $s =_{E^c_{\mathcal{L}}} t$ it guarantees that $q = q'$ in the simplified automaton $\mathcal{S}_E(\mathcal{A})$. By generalising this reasoning, we can show that there exists a subterm $t \in G$ of s such that $s \to^*_{\vec{E^c_{\mathcal{L}}}} t$ and $t \to^*_{\mathcal{S}_E(\mathcal{A})} q$. t is a normal form of G recognised by q. $\mathcal{S}_E(\mathcal{A})$ being 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 recognising a term of $\mathcal{L}$ recognises a term of G. Hence there are no more states in $\mathcal{S}_E(\mathcal{A})$ than terms in G. □

Next, we state and prove the termination theorem, using $E \supseteq E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$. We prove that, at some point, the completed automaton will have the following structure, shown in Figure 1. Each cross represents a term recognised 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 recognised 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.

Theorem 1. *Let $\mathcal{A}$ be an 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. For every index k and state q of $\mathcal{A}^k_{\mathcal{R},E}$, q satisfies at least one of the following properties:

- $P_0(q)$: There exists a transition $f \to q$ in $\mathcal{A}^k_{\mathcal{R},E}$, with $f \in \mathcal{F}^0$

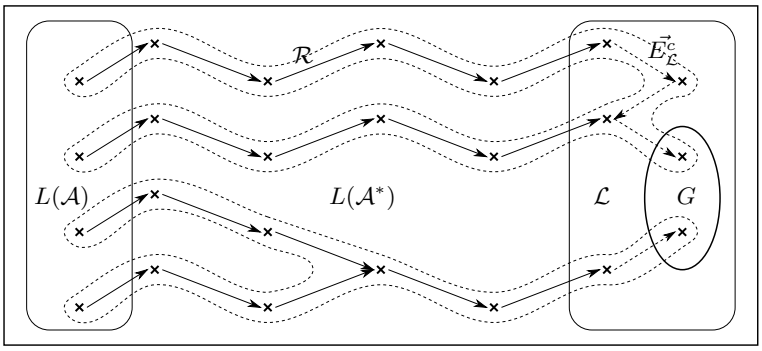


Fig. 1: Completion termination proof idea

– $P_{i+1}(q)$: There exists a transition $f(q_1, \ldots, q_n) \to q$ in $\mathcal{A}^k_{\mathcal{R},E}$ such that $P_i(q_1) \wedge \cdots \wedge P_i(q_n)$.

If not, then $L(\mathcal{A}_{\mathcal{R},E}, q) = \emptyset$ which contradicts the fact that $\mathcal{A}_{\mathcal{R},E}$ is reduced. We know there is a finite number of states recognising terms of $\mathcal{L}$ 2. Let $\tilde{Q}$ be the set of those states. First we prove that for every symbol f and index k such that $\mathcal{A}^k_{\mathcal{R},E}$ contains a transition $f(q_1, \ldots, q_n) \to q$, with $q_1, \ldots, q_n \in \tilde{Q}$, there exists an index k_f such that for all $k' > k_f$, if $\mathcal{A}^{k'}_{\mathcal{R},E}$ contains $f(q_1, \ldots, q_n) \to q'$, then $q' \in \tilde{Q}$. By hypothesis, for any term s such that $s \to^*_{\mathcal{A}^k_{\mathcal{R},E}} f(q_1, \ldots, q_n) \to_{\mathcal{A}^k_{\mathcal{R},E}} q$, there exists a term $t \in \mathcal{L}$ such that $s \to^l_{\mathcal{R}} t$. After at most l successive completion steps, and because $E \supseteq E_{\mathcal{R}}$, we will have q' merged with a state q_f of $\tilde{Q}$. Moreover, having $E \supseteq E_r$ ensures that for every transition $f(q_1, \ldots, q_n) \to q \in \mathcal{A}^k_{\mathcal{R},E}$ such that $k \geq k + l$ we have $q = q_f$.

Let $\tilde{k} = max\{k_f \mid f \in \mathcal{F}\}$. We show, by induction on the property P, that for all $k \geq \tilde{k}$, all states of $\mathcal{A}^k_{\mathcal{R},E}$ are in $\tilde{Q}$. For every q such that $P_0(q)$, there exists a transition $f \to q$ with $f \in \mathcal{F}^0$. Since $k > k_f$, we have shown earlier that $q \in \tilde{Q}$. Assume now that it holds for any state q and index i such that $P_i(q)$. For every q such that $P_{i+1}(q)$, there exists a transition $f(q_1, \ldots, q_n) \to q$ with $P_i(q_1) \wedge \cdots \wedge P_i(q_n)$. By induction hypothesis, $q_1, \ldots, q_n \in \tilde{Q}$. As already shown, this implies $q \in \tilde{Q}$. So, after at most $\tilde{k}$ steps, all states of $\mathcal{A}^k_{\mathcal{R},E}$ are merged into $\tilde{Q}$. Since $\tilde{Q}$ is finite, and $\mathcal{A}^k_{\mathcal{R},E}$ REFD for all k, the algorithm terminates. □

4 A Class of Analysable 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 proves that the completion algorithm terminate on any functional program $\mathcal{R}$. If this works in theory, in practice we want to avoid introducing equations over the application symbol (such as $@(x, y) = y$). Contracting equations on applications makes sense in certain cases, *e.g.*, with idempotent functions ($@(sort, @(sort, x)) = @(sort, x)$), but in most cases, such equations dramatically lower the precision of the completion algorithm.

We want to identify 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 a language $\mathcal{L}$ still has to have a finite number of normal forms w.r.t. $\vec{E^c_{\mathcal{L}}}$, it cannot include terms containing an un-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 $\mathcal{B}^n$ to be the set of terms where such stack size is bounded by n.

Definition 7. *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:*

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

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 @. However for all k, for all term $t \in L_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ there is no term $s \in \mathcal{B}^n \cap IRR(\mathcal{R})$ s.t. $t \to^*_{\mathcal{R}} s$ in general. Theorem 1 cannot be instantiated with $\mathcal{L} = \mathcal{B}^n \cap IRR(\mathcal{R})$. instead 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 instantiate 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.

4.1 Types

In order to define $\mathcal{K}$ and $\mathcal{K}^n$ we require the TRS to be well-typed. Our definition of types is inspired by [1]. Let $\mathscr{A}$ be a non-empty set of *algebraic types*. The set of *types* $\mathscr{T}$ is inductively defined as the least set containing $\mathscr{A}$ 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:

$$ar(A) = 0 \qquad \Leftarrow A \in \mathscr{A}$$
$$ar(A \to B) = 1 + ar(B) \qquad \Leftarrow A \to B \in \mathscr{T}$$

Instead of using alphabets, in a typed terms environment we use *signatures* $\mathcal{F} = \mathcal{C} \cup \mathcal{D} \cup \{@\}$ where $\mathcal{C}$ is a set of *constructor* symbols associated to a unique type $\mathcal{D}$ a set of *defined* symbols of non-zero arity associated to a unique type and @ the application symbol (with no type). Constructor symbols are used to build values, and defined symbols are used to define control structures such as if. We also assign a type to every variable. We write $f : A$ if the symbol f has 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 8 (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.

Definition 9 (Functional TRS). *A higher-order functional TRS 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:*

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

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, all defined functions are total.*

Types provides information 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 $ar(A \to B) = 1$. Furthermore, for certain typed, functional TRSs we can guarantee the absence of partial applications in the result of a computation. For a given signature $\mathcal{F}$, the *order* of a type A, written $ord(A)$, is inductively defined on the structure of A by:

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

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 $ord(int) = 0$ and $ord(int \to int) = 1$.

Example 4. Define two different types of lists $list$ and $list'$. The first defines lists of int with the constructor $consA : int \to list \to list \in \mathcal{C}$, while the second defines lists of functions with the constructor $consB : (int \to int) \to list' \to list' \in \mathcal{C}$. The importance of order becomes manifest here: in the first case a fully reduced term of type $list$ cannot contain any $@$ whereas in the second case it can. $ord(list) = 0$ and $ord(list') = 1$.

Lemma 3. *Let $\mathcal{R}$ be a terminating functional TRS and A a type such that $ord(A) = 0$. Then all terms t of type A are rewritten into a final 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 The class $\mathcal{K}$

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 rules $l \to r \in \mathcal{R}$, 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 applications during rewriting. The definition of $\mathcal{K}$ takes advantage of the type structure and Lemma 3. $\mathcal{K}$ is inductively defined by:

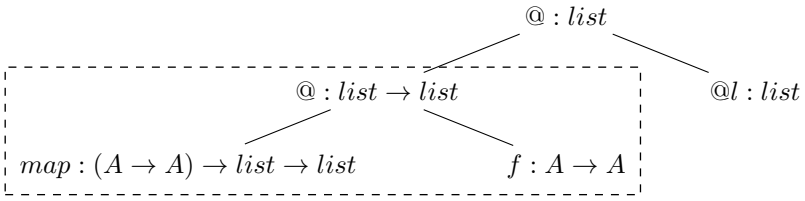
$$\underline{x} : A \in \mathcal{K} \Leftarrow \underline{x} : A \in \mathcal{X}$$
$$f(t_1, \ldots, t_n) : A \in \mathcal{K} \Leftarrow f \in \mathcal{F}^n \setminus \{@\} \wedge t_1, \ldots, t_n \in \mathcal{K} \wedge arity(A) = 0$$
$$f(t_1 : A_1, \ldots, t_n : A_n) : A \in K \Leftarrow f \in \mathcal{F}^n \setminus \{@\} \wedge t_1, \ldots, t_n \in K \wedge ord(A_1) = 0, \ldots, ord(A_n) = 0 \quad (1)$$
$$@(t_1 : A \to B, t_2 : A) : B \in \mathcal{K} \Leftarrow t_1 \in \mathcal{Z}, t_2 \in \mathcal{K} \wedge arity(B) = 0 \quad (2)$$
$$@(t_1 : A \to B, t_2 : A) : B \in \mathcal{K} \Leftarrow t_1, t_2 \in \mathcal{K} \wedge ord(A) = 0 \quad (3)$$

with $\mathcal{Z}$ defined by:

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

The rules (2) and (3) ensure that an application $@(t_1, t_2)$ is either: (2) a total application, and the whole term can be rewritten; or (3) a partial application where t_2 can be rewritten into a term of $\mathcal{B}^0$ (Lemma 3). Rule (1) can be seen as the uncurryfied version of (2). In (2), the purpose of $\mathcal{Z}$ isto allow partial applications inside the total application of a multi-parameter function.

Example 5. Consider the classical *map* function. A typical call to this function is $@(@(map, f), l)$ of type $Alist$, where f is a mapping function, and l a list.



The whole term belongs to $\mathcal{K}$ because of rule (2): $list$ is an algebraic type and its subterm $@(map, f) : list \to list$ belongs to $\mathcal{Z}$. This subterm is a partial application, but there is no risk of stacking partial applications as it is part of a complete call (to the *map* function).

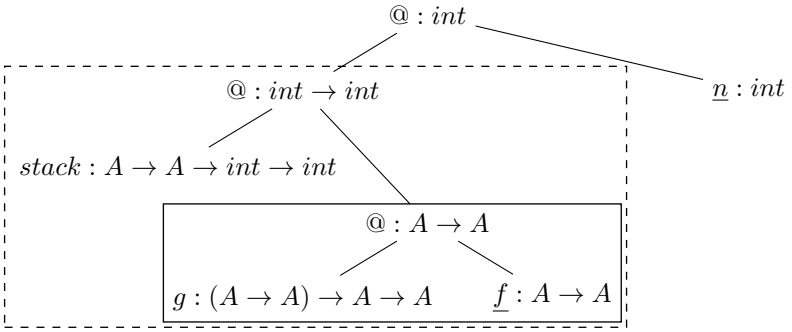
Example 6. Consider the function *stack* defined by:

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

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:

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

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. Notice that $\underline{x}$ is a function and $@(g, \underline{x})$ a partial application. Hence the term $@(@(stack, @(g, \underline{x})), \underline{n})$ is not in $\mathcal{K}$, so the TRS should no be accepted.



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. The idea is the following:

- First we prove that $\mathcal{K}^n_{\trianglerighteq}$ is closed by $\to_{\mathcal{R}}$.
- We prove that if $\mathcal{A}$ recognises terms of $\mathcal{K}^n_{\trianglerighteq}$ it is preserved by completion using the notion of $\mathcal{K}^n$-coherence of $\mathcal{A}$.
- We prove that $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ where B is a fixed upper bound of the arity of all the types of the program.

– Then we prove that there are a finite number of normal form of $\mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ w.r.t $\vec{E^c_{\mathcal{L}}}$.
– Finally we use those three properties combined, and instantiate Theorem 1 with $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ to prove Theorem 2, defined as follows:

Theorem 2. *Let $\mathcal{A}$ be a $\mathcal{K}^n$-coherent REFD tree automaton, $\mathcal{R}$ a 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.*

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 9) 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 $ord(T_1) = 0, \ldots, ord(T_n) = 0$. 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 $ord(T_2) = 0, t_1, t_2 \in \mathcal{K}$. 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 $t = f(t_1, \dots, t_n), f \in \mathcal{F}^n$. 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 $ord(A) = 0, t_1, t_2 \in \mathcal{K}^n$. 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 recognised language stays in $\mathcal{K}^n$, we require the considered automaton to be $\mathcal{K}^n$*-coherent.*

Definition 10 ($\mathcal{K}^n$-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 recognised by all states $q \in \mathcal{Q}$ is $\mathcal{K}^n$-coherent.

If $\mathcal{K}^n$-coherence is not preserved during completion, there is no assurance that no states in the completed automaton can recognise terms outside of $\mathcal{K}^n_\trianglerighteq$. Our goal is to show that it is preserved 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 $s, t \in \mathcal{K}^n_{\trianglerighteq}$ 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 recognised 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 Normalisation 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 (Normalisation 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 $\{f(q_1, \ldots, q_m) \to q'\} = Norm_\Delta(f(q_1, \ldots, q_m) \to q')$, with $f(q_1, \ldots, q_m)$ $\mathcal{K}^n$-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 3 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 12, $\mathcal{A}'$ 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 $E = E^r \cup E^c_n \cup E_\mathcal{R}$. 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. $s =_{E^c_n}$, 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 $E = E^r \cup E^c_n \cup E_\mathcal{R}$ 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 Lemmas:

Lemma 16. *Let $B \in \mathbb{N}$ be the maximum function arity in the program. For all $t : T \in K^n$, $t : T \in IRR(\mathcal{R}) \Rightarrow t : T \in \mathcal{B}^{n+B-arity(T)}(\mathcal{C})$.*

Proof. By induction on $t : T$.

– $t : T \in \mathcal{B}^n(\mathcal{C})$. Since B is the maximum program arity, $arity(T) \leq B$, so $n \leq n + B - arity(T)$. Then $t : T \in \mathcal{B}^{n+B-arity(T)}(\mathcal{C})$.
– $t = f(t_1 : T_1, \ldots, t_n : T_n) : T, arity(T) = 0$. Since $\mathcal{R}$ is a functional TRS, if $t \in IRR(\mathcal{R})$ and $arity(T) = 0$ then $f \in \mathcal{C}^n$. By induction hypothesis, for all $t_i, t_i \in \mathcal{B}^{n+B-arity(T_i)}(\mathcal{C})$, with $\mathcal{B}^{n+B-arity(T_i)}(\mathcal{C}) \subseteq \mathcal{B}^{n+B}(\mathcal{C})$. This implies $t_1, \ldots, t_n \in \mathcal{B}^{n+B}(\mathcal{C})$, so by definition, $t \in \mathcal{B}^{n+B}(\mathcal{C})$. Since $arity(T) = 0$, $t \in \mathcal{B}^{n+B-arity(T)}(\mathcal{C})$.
– $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ with $order(T_1) = 0, \ldots, order(T_n) = 0$. If $arity(T) = 0$ then we can use the same reasoning. Otherwise, for all t_i since $order(T_i) = 0$ we have $t_i \in \mathcal{B}^0(\mathcal{C})$, so $t \in \mathcal{B}^1(\mathcal{C})$. Since $n > 0$, then $B > 0, B > arity(T)$ and $1 \leq n + B - arity(T)$. So by definition, $t \in \mathcal{B}^{n+B-arity(T)}(\mathcal{C})$.
– $t = @(t_1 : T_1, t_2 : T_2)$. Since $\mathcal{R}$ is a functional TRS, if $t \in IRR(\mathcal{R})$ then we can't have $arity(T) = 0$, so by definition of K^n, $order(T_2) = 0$ and $t_1, t_2 \in K^n$. Moreover, since $t_2 \in IRR(\mathcal{R})$, $t_2 \in \mathcal{B}^0(\mathcal{C})$, $t_2 \in \mathcal{W}(\mathcal{C})$. By induction hypothesis, since $t_1 \in IRR(\mathcal{R})$ we have $t_1 \in \mathcal{B}^{n+B-arity(T_2)}(\mathcal{C})$. By definition, $t \in \mathcal{B}^{n+B-arity(T_2)+1}$, and since $arity(T_2) = arity(T) - 1$, $t \in \mathcal{B}^{n+B-arity(T)}$.

Lemma 17. *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. Note that if $t \in \mathcal{K}^n_\trianglerighteq$ then $t \in \mathcal{Z}^n$. We reason by induction on $t : T$.

– $t : T \in \mathcal{K}^n$, then by Lemma 16 we have $t : T \in \mathcal{B}^{n+B-arity(T)}(\mathcal{C})$, thus $t : T \in \mathcal{B}^{n+2B-arity(T)}(\mathcal{C})$.
– $t = @(t_1 : T_1, t_2 : T_2) : T$ with $t_1 \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$. By Lemma 16 we have $t_2 : T_2 \in \mathcal{B}^{n+B}(\mathcal{C})$. By induction hypothesis we have $t_1 : T_1 \in \mathcal{B}^{n+2B-arity(T_2)}(\mathcal{C})$. Since $arity(T_2) = arity(T) + 1$, $t_1 : T_1 \in \mathcal{B}^{n+2B-arity(T)-1}(\mathcal{C})$.
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 $t : T \in \mathcal{B}^{max(n+2B-arity(T),n+B+1)}(\mathcal{C})$.
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 $t : T \in \mathcal{B}^{n+2B-arity(T)}(\mathcal{C})$.

4.3 Proof of Theorem 2

Proof. 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 $IRR(\mathcal{R}) \cap \mathcal{K}^n_{\trianglerighteq} \subseteq \mathcal{B}^{n+2B}$ (Lemma 17). Let $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. $\mathcal{R}$ is terminating, so for every term $s \in \mathcal{L}_{\trianglerighteq}(\mathcal{A}^k_{\mathcal{R},E})$ there exists $t \in \mathcal{L}$ such that $s \to^*_{\mathcal{R}} t$. Since the number of normal form of $\mathcal{L}$ is finite w.r.t $\vec{E}$, Theorem 1 implies that the completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates.

5 Equation Generation

Theorem 2 states a number of hypotheses that must be satisfied in order to guarantee termination of the completion algorithm:

- We expect the initial automaton $\mathcal{A}$ to be $\mathcal{K}^n$-coherent and REFD.
- We shall keep the assumption that $\mathcal{R}$ is terminating, and not address the termination of $\mathcal{R}$.
- All left-hand sides of rules of $\mathcal{R}$ are in the set of terms $\mathcal{K}$. This is a straightforward syntactic check. If it is not verified, we reject the TRS before starting the completion.
- The set of equations E must be of the form $E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$. The equation sets E^r and $E_{\mathcal{R}}$ are determined directly from the syntactic structure of $\mathcal{R}$. However, there is no unique suitable set of contracting equations $E^c_{\mathcal{L}}$. This set must be generated carefully, because a bad choice of contracting equations (*i.e.*, equations that equate too many terms) will have a severe negative impact on the precision of the analysis result.

In this section, we describe a method for generating all possible sets of contracting equations $E^c_{\mathcal{L}}$. To simplify the presentation, we only present the case where $\mathcal{L} = \mathcal{W}(\mathcal{C})$ and $IRR(\mathcal{R}) \subseteq \mathcal{W}(\mathcal{C})$ (*i.e.*, all results are first-order). Our approach looks for contracting equations for the set of ground terms $\mathcal{W}(\mathcal{C})$ instead of the set $\mathcal{B}^{n+2B}$ mentioned in Theorem 2. More precisely, we generate the set of equations iteratively, as a series of equation sets $\mathbb{E}^k_c$ where the equations only equate terms of depth at most k. Recall that a contracting equation is of the form $u = u|_p$ with $p \neq \lambda$, *i.e.*, it equates a term with a strict subterm of the same type. A set of contracting equations over the set $\mathcal{W}(\mathcal{C})$ is then generated as follows: (i) generate the set of left-hand side of equations as a *covering set of terms* [23], so that for each term $t \in \mathcal{W}(\mathcal{C})$ there exists a left-hand side u of an equation and a substitution σ such that $t = u\sigma$. (ii) for each left-hand side, generate all possible equations of the form $u = u|_p$, satisfying that both sides have the same type. (iii) from all those equations, we build all possible $E^c_{\mathcal{L}}$ (with $\mathcal{L} = \mathcal{W}(\mathcal{C})$) such that the set of normal forms of $\mathcal{W}(\mathcal{C})$ w.r.t. $\vec{E^c_{\mathcal{L}}}$ is finite. Since $\vec{E^c_{\mathcal{L}}}$ is left-linear and $\mathcal{L} = \mathcal{W}(\mathcal{C})$, this can be decided efficiently [11].

Example 7. Assume that $\mathcal{C} = \{s, 0\}$ where s has arity 1 and 0 has arity 0. For $k = 1$, the covering set is $\{s(x), 0\}$ and $\mathbb{E}^1_c = \{\{s(x) = x\}\}$. For depth 2, covering set is $\{s(s(x)), s(0), 0\}$ and $\mathbb{E}^2_c = \mathbb{E}^1_c \cup \{\{s(s(x)) = x\}, \{s(s(x)) = s(x)\}, \{s(0) = 0\}, \{s(0) = 0, s(s(x)) = x\}, \{s(0) = 0, s(s(x)) = s(x)\}\}$. All equation sets of $\mathbb{E}^k_c$ and $\mathbb{E}^2_c$ satisfy Definition 6 and lead to different approximations.

To find every left-hand side we use the notion of *covering set of terms*, inspired by [23]. 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 $u \in S$, $depth(u) \leq depth(t) + k$ where $depth(u)$ is defined in [1]. From CT_k we can compute the set $\mathbb{E}^k_c$ of all the possible sets of contracting equations, where for each equation $u = u|_p$, $depth(u) \leq k + 1$.

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

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.

5.1 Verifying a program

To verify a property φ on a program, we use completion and equation generation as follows. The program is represented by a TRS $\mathcal{R}$ and function calls are represented by an initial tree automaton $\mathcal{A}$. Both have to respect the hypothesis of Theorem 2. 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 $L(\mathcal{A}_k^*)$ *contains a counter example* **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 $L(\mathcal{A}_k^*)$ *contains no counter example* **then**
        $\mathcal{A}^* \leftarrow Completion_{\mathcal{R},E}(\mathcal{A})$;
        if $L(\mathcal{A}^*)$ *contains no counter example* **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;

The algorithm searches for a set of contracting equations E_c such that verification succeeds, *i.e.* $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$ satisfy φ. Starting from $k = 1$, we apply the following algorithm:

1. We first complete the tree automaton $\mathcal{A}^k$ recognising the *finite* subset of $\mathcal{L}(\mathcal{A})$ of terms of maximum depth k. Since $\mathcal{L}(\mathcal{A}^k)$ is finite and $\mathcal{R}$ is terminating, the set of reachable terms is finite, completion terminates without equations and computes an automaton $\mathcal{A}^{k*}_{\mathcal{R}}$ recognising exactly the set $\mathcal{R}^*(L(\mathcal{A}^k))$ [17].
2. If $\mathcal{L}(\mathcal{A}^{k*}_{\mathcal{R}})$ does not satisfy φ then verification fails: a counterexample is found.
3. Otherwise, we search for a suitable set E_c. All E_c of $\mathbb{E}_c^k$ that introduce a counterexample in the completion of $\mathcal{A}^k$ with $\mathcal{R}$ and E_c are filtered out.
4. Otherwise for all remaining E_c, we apply try to complete $\mathcal{A}$ with $\mathcal{R}$ and $E = E_r \cup E_{\mathcal{R}} \cup E_c$ and check φ on the completed automaton. If φ is true on $\mathcal{A}^*_{\mathcal{R},E}$ then verification succeeds. Otherwise, we try the next E_c.
5. If there remain no E_c, we start again with $k = k + 1$.

If there exists a set of equations E_c able to verify the program, this algorithm will find it eventually, or find a counter example. However if there is no set of equations that can verify the program, this algorithm does not terminate.

6 Experiments

The verification technique described above has been integrated in the Timbuk library [16]. We implemented the naive equation generation where all possible equation sets E_c are enumerated. Despite the evident scalability issues of this simple-minded version of the verification algorithm, we have been able to verify a series of properties of several classical higher-order functions: *map*, *filter*, *exists*, *forall*, $foldRight$, $foldLeft$ as well as higher-order sorting functions parameterised by an ordering function. Most examples are taken from or inspired by [26,24] and have corresponding TRSs in the $\mathcal{K}$ class defined above. Given $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{A}^*_{\mathcal{R},E}$, the correctness of the verification, i.e. the fact that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, can be certified using a formally verified

checker [6,14]. This permits to reuse the corresponding verification result in a Coq or an Isabelle/HOL proof. All our (successful) completion attempts output a `comp.res` file, containing $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{A}^*_{\mathcal{R},E}$, which can be certified automatically using the checker of [6]. Note that $\mathcal{A}$ is transformed to be REFD and $\mathcal{R}/E$-coherent [18]. The precision theorem of [18] gives us an additionnal crucial precision property: $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \subseteq (\mathcal{R}/E)^*(\mathcal{L}(\mathcal{A}))$, i.e. $\mathcal{A}^*_{\mathcal{R},E}$ recognises no more terms than terms reachable by rewriting with $\mathcal{R}$ modulo equations of E. In other words, completion introduce no more approximation than what E defines. 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` is the automaton defining the set of (typed) constructor terms where each state recognises a type. This is necessary input 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 FinalStates 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 FinalStates 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 recognised 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. Timbuk's site http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/ lists those verification experiments. Nine of them are automatically proven. Two other examples show that correct counter-examples are generated when the property is not provable. On one example equation generation times out due to our naïve enumeration of equations. For this last case, by providing the right set of equations in `mapTree2NoGen` the verification of the function succeeds.

7 Related Work

When it comes to verifying first-order imperative programs, there exist several successful tools based on abstract interpretation such as ASTREE [3] and SLAM [2]. The use of abstract interpretation for verifying higher-order functional programs has comparatively received less attention. The tree automaton completion technique is one analysis technique able to verify first-order Java programs [4]. Until now, the completion algorithm was guaranteed to terminate only in the case of first-order functional programs [17].

Liquid Types [29], followed by Bounded Rafinement Types [33,32], and also Set-Theoretic Types [9,8], are all attempts to enrich the type system of functional languages to prove non-trivial properties on higher-order programs. However, these methods are not automatic. The user has to express the property he wants to prove using the type system, which can tedious and/or difficult. In some cases, the user even has to specify straightforward intermediate lemmas to help the type checker.

The first attempt in verifying regular properties came with Jones [19] and Jones and Andersen [20]. Their technique computes a grammar over-approximating the set of states reachable by a rewriting systems. However, their approximation is fixed and too rough to prove programs like Example 1 (*filter even*). Our program and property models are close to those of Jones and Andersen. However, the approximation in our analysis is not fixed and can be automatically adapted to the verification objective.

Ong *et al.* proposes one way of addressing the precision issue of Jones and Andersen's approach using a model checking technique on Pattern Matching Recursion Schemes [26,31] (PMRS). This technique improves the precision but is still not able to verify functions such as Example 1 (see [30] page 85). As shown in our experiments, our technique handles this example.

Kobayashi *et al.* developed a tree automata-based technique [24] (but not relying on TRS and completion), able to verify regular properties (including safety properties on Example 1). We have verified a selection of examples coming from [24] and observed that we can verify the same regular properties as they can. Our prototype implementation is inferior in terms of execution time, due to the slow generation of equations. A strength of our approach is that our verification results are certifiable (see Section 6), and can be transformed into a formal Coq proof that the program satisfies the property.

Our verification framework is based on regular abstractions and uses a simple abstraction mechanism based on equations. This particular combination provides a sweet spot, even though more powerful abstraction mechanisms have been described in the literature. Regular abstractions are less expressive than Higher-Order Recursion Schemes [27,21] or Collapsible Pushdown Automata [7], and equation-based abstractions are a particular case of predicate abstraction [22]. However, the two restrictions imposed in this particular framework results in two strong benefits. First, the precision of the approximation is formally defined and precisely controlled using equations: $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \subseteq (\mathcal{R}/E)^*(\mathcal{L}(\mathcal{A}))$ (see Section 6). This precision property permits to prove intricate properties with simple (regular) abstractions. Second, using tree automata-based models eases the certification of the verification results in a proof assistant. This significantly increases the confidence in the verification result compared to verdicts obtained by complex CEGAR-based model-checkers whose implementation is error-prone.

8 Conclusion & Future Work

This paper shows that tree automata completion is a simple yet powerful, fully automatic verification technique for higher-order functional programs, expressed as term rewriting systems. We have proved that the completion algorithm terminates on a class of TRS encompassing common functional programs, and provided experimental evidence of the viability of the approach by verifying properties on fundamental higher-order functions including filtering and sorting.

One remaining theoretical question is whether this approach is complete, i.e. if there exists a regular approximation of the reachable terms of a functional program, can we build it using equations? For general rewriting, when the set of reachable terms is regular, we already know that the answer is positive using contracting equations defined on $\mathcal{T}(\mathcal{F})$ [15]. Extending this result to approximations and higher-order functional programs is a promising research topic.

The generation of the approximating equations is automatic but simple-minded, and too simple to turn the prototype into a full verification tool. Further work should look into how sets of contracting equations can be generated in a more efficient manner, notably by taking the structure of the TRS into account and using a CEGAR approach.

The present verification technique is agnostic to the evaluation strategy. An interesting research track would be to experiment completion-based verification techniques with different term rewriting semantics of functional programs such as outlined by Clemente *et al.* [10]. This would permit to take a particular evaluation strategy into account, and in certain cases simplify the verification. This is in line with our long-term research goal of providing a light-weight verification tool to assist the working OCaml programmer.

Our work focuses on verifying regular properties represented by tree automata. Dealing with non-regular over-approximations of reachable terms would allow us to verify relational properties like comparing the length of the list before and after $filter$. This is one of the objective of techniques like [22]. Building non-regular over-approximations of reachable terms for TRS, using a form of completion, is possible [5]. However, up to now, adapting automatically the precision of such approximations to a given verification goal is not possible. Extending their approach with equations may provide a powerful verification tool worth pursuing.

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., Genet, T., Jensen, T., Leroux, L.: Rewriting Approximations for Fast Prototyping of Static Analyzers. In: RTA'07. Volume 4533 of LNCS., Springer (2007) 48–62
5. Boichut, Y., Chabin, J., Réty, P.: Towards more precise rewriting approximations. In: LATA'15. Volume 8977 of LNCS., Springer (2015) 652–663
6. Boyer, B., Genet, T., Jensen, T.: Certifying a Tree Automata Completion Checker. In: IJCAR'08. Volume 5195 of LNCS., Springer (2008)
7. Broadbent, C.H., Carayol, A., Hague, M., Serre, O.: C-shore: a collapsible approach to higher-order verification. In: ICFP'13, ACM (2013)
8. 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)
9. 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)
10. Clemente, L., Parys, P., Salvati, S., Walukiewicz, I.: Ordered tree-pushdown systems. In: FSTTCS'15. Volume 45 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015) 163–177
11. Comon, H.: Sequentiality, Monadic Second-Order Logic and Tree Automata. Inf. Comput. **157**(1-2) (2000) 25–51
12. 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)
13. Coq: The coq proof assistant reference manual: Version 8.6. (2016)
14. Felgenhauer, B., Thiemann, R.: Reachability, confluence, and termination analysis with state-compatible automata. Inf. Comput. **253** (2017) 467–483
15. Genet, T.: Automata Completion and Regularity Preservation. Technical report, Inria (2017) https://hal.inria.fr/hal-01501744.
16. 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/.
17. Genet, T.: Termination criteria for tree automata completion. Journal of Logical and Algebraic Methods in Programming **85**(1) (2016) 3–33
18. Genet, T., Rusu, V.: Equational approximations for tree automata completion. Journal of Symbolic Computation **45**(5) (2010) 574–597
19. Jones, N.D.: Flow analysis of lazy higher-order functional programs. In Abramsky, S., Hankin, C., eds.: Abstract Interpretation of Declarative Languages. Ellis Horwood, Chichester, England (1987) 103–122

20. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theoretical Computer Science **375**(1-3) (2007) 120–136
21. 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
22. 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
23. Kounalis, E.: Testing for the ground (co-)reducibility property in term-rewriting systems. Theor. Comput. Sci. **106**(1) (1992) 87–117
24. Matsumoto, Y., Kobayashi, N., Unno, H.: Automata-based abstraction for automated verification of higher-order tree-processing programs. In: Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings. (2015) 295–312
25. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. TCS **403**(2-3) (2008) 239–264
26. 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
27. 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
28. Paulson, L.C., et al.: The isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory (1993)
29. 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
30. Salmon, Y.: Analyse d'atteignabilité pour les programmes fonctionnels avec stratégie d'évaluation en profondeur. Phd thesis, University of Rennes 1 (2015)
31. 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
32. 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
33. 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