Verifying Higher-Order Functions with Tree Automata

Thomas Genet, Timothée Haudebourg, Thomas Jensen

Verifying Higher-Order Functions with Tree Automata

Thomas Genet, Timothée Haudebourg, and Thomas Jensen

Univ. Rennes, INRIA, IRISA

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.

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 [30], or write a specification in a formalism such as Liquid Types [31] or Bounded Refinement Types [34,33] 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 [21] 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.* [29] and Kobayashi [23,24,26] 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) [20], 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. Moreover, our completion-based approach 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.

$$\begin{aligned}
@(@(\mathit{filter}, \underline{p}), \mathit{cons}(\underline{x}, \underline{l})) &\to \textbf{if } @(\underline{p}, \underline{x}) \textbf{ then } \mathit{cons}(\underline{x}, @(@(\mathit{filter}, \underline{p}), \underline{l})) \\
&\qquad\qquad \textbf{else } @(@(\mathit{filter}, \underline{p}), \underline{l}) \\
@(@(\mathit{filter}, \underline{p}), \mathit{nil}) &\to \mathit{nil} \\
@(\mathit{even}, 0) &\to \mathit{true} \qquad\qquad @(\mathit{even}, s(\underline{x})) \to @(\mathit{odd}, \underline{x}) \\
@(\mathit{odd}, 0) &\to \mathit{false} \qquad\qquad @(\mathit{odd}, s(\underline{x})) \to @(\mathit{even}, \underline{x})
\end{aligned}$$

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 $\mathcal{R}$ with 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* does not belong to the regular set $L(\mathcal{A}^*)$. We denote by *regular properties* the family of properties characterised by a regular set. In particular, regular properties do not count symbols in terms, nor relate subterm heights (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 [19]. 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 by introducing an approximation based on *equational abstraction* [27]: $L(\mathcal{A}^*) \supseteq \mathcal{R}^*(L(\mathcal{A}))$. Equations make TAC powerful enough to verify first-order functional programs [19]. 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 covers common 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).

All proofs missing in this paper can be found in the accompanying technical report [17]. The paper is organised as follow: 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 the certification of results in Coq. We present related work in Section 7. Section 8 concludes the paper.

2 Background

This section introduces basic concepts used throughout the paper. We recall the usual definitions of term rewriting systems and tree automata, and present the completion algorithm which forms the basis of our verification technique.

2.1 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. It is 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. We write $s \trianglerighteq t$ if t is a subterm of s and $s \triangleright t$ if it is a subterm and $s \neq t$. If $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$, 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.

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

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

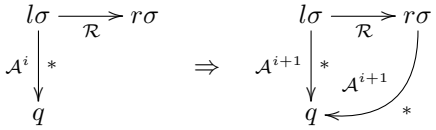
The term $cons(s(0), nil)$ is *irreducible* (no rule applies to it) and hence the result of the function call. We write $IRR(\mathcal{R})$ for the set of irreducible terms of $\mathcal{R}$.

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 $\mathcal{F} \cup \mathcal{Q}$. Rules in Δ, called *transitions*, are 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 $t \to^*_{\Delta} q$, which we also write $t \to^*_{\mathcal{A}} q$. 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 $q \in \mathcal{Q}_f$ s.t. $t \in L(\mathcal{A}, q)$. 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 $\mathcal{F} = \{0 : 0, s : 1\}$, $\mathcal{Q}_f = \{q_{pair}\}$ and $\Delta = \{0 \to q_{pair}, s(q_{odd}) \to q_{pair}, s(q_{pair}) \to q_{odd}, nil \to q_{list}, cons(q_{pair}, q_{list}) \to q_{list}\}$ recognises all lists of even natural numbers.

An *ϵ-transition* is a transition $q \to q'$ where $q \in \mathcal{Q}$. A tree automaton $\mathcal{A}$ is *ϵ-free* if it contains no ϵ-transitions. $\mathcal{A}$ 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 q there is at least one term t such that $t \to^*_{\Delta} q$.

2.2 Tree Automata Completion algorithm

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 $\mathcal{A}^0$, 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 set $\mathcal{R}^*(L(\mathcal{A})) = \{t \mid \exists s \in L(\mathcal{A}), s \to_{\mathcal{R}}^* t\}$. It includes all intermediate computations and, in particular, the *output* of the functional program. The algorithm proceeds by computing iteratively $\mathcal{A}^1, \mathcal{A}^2, \ldots$ such that $\mathcal{A}^{i+1} = \mathcal{C}_{\mathcal{R}}(\mathcal{A}^i)$ until it reaches a fix-point, $\mathcal{A}^*$. Here, $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i)$ represents *one step* of completion and is performed by searching and *completing* the *critical pairs* of $\mathcal{A}^i$.



Definition 1 (Critical pair). *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 $l\sigma \to_{\mathcal{A}^i}^* q$ and $r\sigma \not\to_{\mathcal{A}^i}^* q$.*

Completing a critical pair consists in adding the necessary transitions in $\mathcal{A}^{i+1}$ to have $r\sigma \to_{\mathcal{A}^{i+1}}^* q$, and hence $r\sigma \in L(\mathcal{A}^{i+1}, q)$.

Example 2. Let $\mathcal{A}^0$ be the previously defined tree automaton recognising all lists of even natural numbers. Let $\mathcal{R} = \{s(s(x)) \to s(x)\}$. $\mathcal{A}^0$ has a critical pair $\langle s(s(x)) \to s(x), \sigma, q_{pair}\rangle$ with $\sigma(x) = q_{pair}$. To *complete* the automaton, we need to add transition such that $s(q_{pair}) \to_{\mathcal{A}^1}^* q_{pair}$. Since we already have the state q_{odd} recognising $s(q_{pair})$, we only add the transition $q_{odd} \to q_{pair}$. The formal definition of the completion step, including the procedure of choosing which new transition to introduce, can be found in [17].

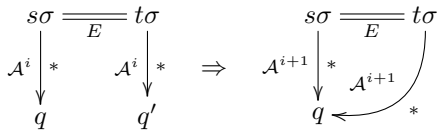
Every completion step has the following property:

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

It implies that, if a fix-point $\mathcal{A}^*$ then it recognises every term of $\mathcal{R}^*(L(\mathcal{A}))$. However it is in general impossible to compute a tree automaton recognising $\mathcal{R}^*(L(\mathcal{A}))$ exactly, and this may cause the completion algorithm to diverge. 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 by introducing a set E of *equations* of the form $l = r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. From E we derive the relation $=_E$, the *smallest congruence* such that for all equation $l = r$ and substitution σ we have $l\sigma =_E r\sigma$. In this paper we also write $\vec{E}$ for the TRS $\{l \to r \mid l = r \in E\}$. At each completion step, the algorithm *simplifies* the automaton by merging states together according to E.

Definition 2 (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. If $s = t \in E, \sigma : \mathcal{X} \mapsto \mathcal{Q}, q, q' \in \mathcal{Q}$ such that $s\sigma \to_{\mathcal{A}}^* q, t\sigma \to_{\mathcal{A}}^* q'$ and $q \neq q'$ then $\mathcal{A}$ can be simplified into $\mathcal{A}' = \mathcal{A}\{q' \mapsto q\}$ (where q' has been substitued by q), denoted by $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$.*

We write $\mathcal{S}_E(\mathcal{A})$ for the unique automaton (up to renaming) $\mathcal{A}'$ such that $\mathcal{A} \rightsquigarrow_E^* \mathcal{A}'$ and $\mathcal{A}'$ is irreducible by $\rightsquigarrow_E$. One completion step is now defined by $\mathcal{A}^{i+1} = \mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i))$.



Example 3. This example shows how using equations can lead to approximations in tree automata. Let $\mathcal{A}$ be the tree automaton defined by the set of transitions $\Delta = \{0 \to q_0, s(q_0) \to q_1\}$. This automaton recognises the two terms 0 in q_0 and $s(0)$ (also known as 1) in q_1. Let $E = \{s(\underline{x}) = \underline{x}\}$ containing the equation that equates a number and its successor. For $\sigma = \{\underline{x} \mapsto 0\}$ we have $s(\underline{x})\sigma \to_{\mathcal{A}} q_1$, $\underline{x}\sigma \to_{\mathcal{A}} q_0$ and $s(\underline{x})\sigma =_E \underline{x}\sigma$. Then in $\mathcal{S}_E(\mathcal{A})$, q_0 and q_1 are merged. The resulting automaton has transitions $\{0 \to q_0, s(q_0) \to q_0\}$, which recognises $\mathbb{N}$ in q_0.

The idea behind the simplification is to overapproximate $\mathcal{R}^*(L(\mathcal{A}))$ when it is *not regular*. It has been shown in [19] that it is possible to tune the precision of the approximation. 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. Our contribution in this paper consists in finding a class of TRS/programs and equations E for which the completion algorithm with equations terminates.

3 Termination of Tree Automata Completion

In this section, we show that termination of the completion algorithm with a set of equations E is ensured under the following conditions: if (i) $\mathcal{A}^k$ is reduced ϵ-free and deterministic (written **REFD** in the rest of the paper) for all k; (ii) every term of $\mathcal{A}^k$ can be rewritten into a term of a given language $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ using $\mathcal{R}$ (for instance if $\mathcal{R}$ is terminating); (iii) $\mathcal{L}$ has a finite number of equivalence classes w.r.t E. Completion is known to preserve $\not\epsilon$-reduceness and $\not\epsilon$-determinism if $E \supseteq E_r \cup E_{\mathcal{R}}$ [19] where $E_{\mathcal{R}} = \{s = t \mid s \to t \in \mathcal{R}\}$ and $E_r = \{f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \mid f \in \mathcal{F}^n\}$. Condition (i) is ensured by showing that, in our verification setting, completion preserve REFD. The last condition is ensured by having $E \supseteq E_{\mathcal{L}}^c$ where $E_{\mathcal{L}}^c$ is a set of *contracting equations*.

Definition 3 (Contracting Equations). *Let $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$. A set of equations is contracting for $\mathcal{L}$, denoted by $E_{\mathcal{L}}^c$, if all equations of $E_{\mathcal{L}}^c$ 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 4. Assume that $\mathcal{F} = \{0 : 0, s : 1\}$. The set $E^c_{\mathcal{L}} = \{s(x) = x\}$ is contracting for $\mathcal{L} = \mathcal{T}(\mathcal{F})$ because the set of normal forms of $\mathcal{T}(\mathcal{F})$ with respect to $\vec{E^c_{\mathcal{L}}} = \{s(x) \to x\}$ is the (finite) set $\{0\}$. The set $E^c_{\mathcal{L}} = \{s(s(x)) = x\}$ is contracting because the 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. To prove termination of completion, we first 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 $\mathcal{L}$ with respect to $\vec{E^c_{\mathcal{L}}}$. In our case $\mathcal{L}$ will be the set of output terms of the program. Since $\mathcal{A}^*$ does not only recognises the output terms, we need additional states to recognise intermediate computation terms. In the proof of Theorem 1 we show 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 we can show that $\mathcal{A}^*$ is finite.

Theorem 1. *Let $\mathcal{A}$ be an REFD tree automaton, $\mathcal{R}$ a left-linear TRS, E a set of equations and $\mathcal{L}$ a language closed by subterms such that for all $k \in \mathbb{N}$ and for all $s \in L_{\trianglerighteq}(\mathcal{A}^k)$, 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 with a REFD $\mathcal{A}^*$.*

4 A Class of Analysable Programs

The next step is to identify a class of functional programs and a language $\mathcal{L}$ for which Theorem 1 applies. 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 terminates 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. Hence, we want to identify a language $\mathcal{L}$ 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. $\vec{E^c_{\mathcal{L}}}$ (Theorem 1), 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. $@(f, s(@(f, s(@(f, s(x))))))$). To do that we (i) define a set $\mathcal{B}^d$ of all terms where such stack size is bounded by $d \in \mathbb{N}$; (ii) define a set $\mathcal{K}^n$ and a class of TRS called $\mathcal{K}$-TRS such that for any TRS $\mathcal{R}$ in this class, $\mathcal{K}^n$ is closed by $\mathcal{R}$ and $\mathcal{K}^n \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{\phi(n)}$. This is done by first introducing a type system over the terms; (iii) finally define $\mathcal{L} = \mathcal{B}^{\phi(n)} \cap IRR(\mathcal{R})$ that can be used to instantiate Theorem 1.

Definition 4. *For a given alphabet $\mathcal{F} = \mathcal{C} \cup \{@\}$, $\mathcal{B}^d$ is the set of terms where every application depth is bounded by d. It is the smallest set defined by:*

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

In Section 5, we show how to produce E^c such that $\mathcal{B}^d \cap IRR(\mathcal{R})$ has a finite number of normal forms w.r.t. $\vec{E^c}$ with no equations on @. However we don't have for all k, for all term $t \in L_{\trianglerighteq}(\mathcal{A}^k)$ a term $s \in \mathcal{B}^d \cap IRR(\mathcal{R})$ s.t. $t \to_{\mathcal{R}}^* s$ in general. Theorem 1 cannot be instantiated with $\mathcal{L} = \mathcal{B}^d \cap IRR(\mathcal{R})$. Instead we define (i) a set $\mathcal{K}^n \subseteq \mathcal{T}(\mathcal{F})$ and ϕ such that $\mathcal{K}^n \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{\phi(d)}$ and (ii) a class of TRS, called $\mathcal{K}$-TRS for which $L_{\trianglerighteq}(\mathcal{A}^k) \subseteq \mathcal{K}^n_{\trianglerighteq}$. In $\mathcal{K}$-TRS, the right hand sides of TRS rules are contained in a set $\mathcal{K}$ whose purpose is to forbid the construction of unbounded partial applications during rewriting. If the initial automaton satisfies $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n_{\trianglerighteq}$ then we can instantiate Theorem 1 with $\mathcal{L} = \mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R})$ and prove termination.

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:

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

Instead of using alphabets, in a typed terms environment we use *signatures* $\mathcal{F} = \mathcal{C} \cup \{@\}$ where $\mathcal{C}$ is a set of *constructor* symbols associated to a unique type and @ the application symbol (with no type). We also assign a type to every 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. We write $\mathcal{W}(\mathcal{F}, \mathcal{X})$ for the set of all *well typed terms* using the usual definition. We extend the definition of term rewriting systems to typed TRS. A TRS is well typed if all rules are of the form $l : A \to r : A$ (type is preserved). 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 5 (Functional TRS). *A higher-order functional TRS is composed of rules of the form*

$$@(\ldots @(f, t_1 : A_1) \ldots, t_n : A_n) : A \to r : A$$

where $f : A_1 \to \ldots \to A_n \to A \in \mathcal{C}^n$, $t_1 \ldots t_n \in \mathcal{W}(\mathcal{C}, \mathcal{X})$ *and* $r \in \mathcal{W}(\mathcal{F}, \mathcal{X})$. *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 types, we can guarantee the absence of partial applications in the result of a computation using the type's *order*. For a given signature $\mathcal{F}$, the *order* of a type A, written $ord(A)$, is inductively defined on the structure of A by:

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

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

Example 5. 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 1. *If* $\mathcal{R}$ *is a complete functional TRS and* A *a type such that* $ord(A) = 0$, *then all* closed *terms* t *of type* A *are rewritten into an irreducible term with no partial application:*

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

4.2 The class $\mathcal{K}$-TRS

Recall that we want to define (i) a set $\mathcal{K}^n \subseteq \mathcal{T}(\mathcal{F})$ and ϕ such that $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{\phi(n)}$ and (ii) a class of TRS $\mathcal{K}$-TRS for which $L_{\trianglerighteq}(\mathcal{A}^k) \subseteq \mathcal{K}^n_{\trianglerighteq}$. Assuming that $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n_{\trianglerighteq}$ we instantiate Theorem 1 with $\mathcal{L} = \mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R})$ and prove termination.

Definition 6 ($\mathcal{K}$-TRS). *A TRS* $\mathcal{R}$ *is part of* $\mathcal{K}$*-TRS if for all rules* $l \to r \in \mathcal{R}$, $r \in \mathcal{K}$ *where* $\mathcal{K}$ *is inductively defined by:*

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

with $\mathcal{Z}$ *defined by:*

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

By constraining the form of the right hand side of each rule of $\mathcal{R}$, $\mathcal{K}$ defines a set of TRS that cannot construct unbounded partial applications during rewriting. The definition of $\mathcal{K}$ takes advantage of the type structure and Lemma 1. The rules (1) and (2) ensure that an application $@(t_1, t_2)$ is either: (1) a total application, and the whole term can be rewritten; or (2) a partial application where t_2 can be rewritten into a term of $\mathcal{B}^0$ (Lemma 1). In (1), $\mathcal{Z}$ allows partial applications inside the total application of a multi-parameter function.

Example 6. Consider the classical *map* function. A typical call to this function is $@(@(map, f), l)$ of type *list*, where f is a mapping function, and l a list. The whole term belongs to $\mathcal{K}$ because of rule (1): *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 7. Consider the function *stack* defined by:

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

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

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

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 does not belong to the $\mathcal{K}$-TRS class.

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 with Theorem 1 we can prove that the completion of $\mathcal{A}$ with $\mathcal{R}$ terminates. The idea is the following:

- Prove that if $\mathcal{A}$ recognises terms of $\mathcal{K}^n_{\trianglerighteq}$, then it is preserved by completion using the notion of $\mathcal{K}^n$-coherence of $\mathcal{A}$.
- Prove that $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ where $B \in \mathbb{N}$ is a fixed upper bound of the arity of all the types of the program.
- Prove that there is 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 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 7 ($\mathcal{K}^n$-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, then some states in the completed automaton may recognise terms outside of $\mathcal{K}^n_{\trianglerighteq}$. Our goal is to show that it is preserved by $\mathcal{C}_\mathcal{R}(\cdot)$ (Lemma 2) then by $\mathcal{S}_E(\cdot)$ (Lemma 3).

Lemma 2 ($\mathcal{C}_\mathcal{R}(\mathcal{A})$ 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.*

Lemma 3 ($\mathcal{S}_E(\mathcal{A})$ preserves $\mathcal{K}^n$-coherence). *Let $\mathcal{A}$ be a REFD tree automaton, $\mathcal{R}$ a functional TRS and E a set of equations such that $E = E^r \cup E^c_\mathcal{L} \cup E_\mathcal{R}$ with $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent then $\mathcal{S}_E(\mathcal{A})$ is $\mathcal{K}^n$-coherent.*

By using Lemma 2 and Lemma 3, 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. The proofs of these two lemmas are based on a detailed analysis of the completion algorithm itself. The complete proofs are provided in [17].

Lemma 4 (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 $E = E^r \cup E^c_\mathcal{L} \cup E_\mathcal{R}$ with $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ and $\mathcal{A}$ is $\mathcal{K}^n$-coherent then for all $k \in \mathbb{N}$, $\mathcal{A}^k$ is $\mathcal{K}^n$-coherent. In particular, $\mathcal{A}^*$ is $\mathcal{K}^n$-coherent.*

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

Lemma 5. *For all $t : T \in \mathcal{K}^n_{\trianglerighteq}$, $t : T \in IRR(\mathcal{R}) \Rightarrow t : T \in \mathcal{B}^{n+2B-arity(T)}$.*

4.3 Proof of Theorem 2

Proof. According to Lemma 4, for all $k \in \mathbb{N}$, the completed automaton $\mathcal{A}^k$ is $\mathcal{K}^n$-coherent. By definition this implies that $\mathcal{L}_{\trianglerighteq}(\mathcal{A}^k) \subseteq \mathcal{K}^n_{\trianglerighteq}$. Moreover, we know that $IRR(\mathcal{R}) \cap \mathcal{K}^n_{\trianglerighteq} \subseteq \mathcal{B}^{n+2B}$ (Lemma 5). Let $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. $\mathcal{R}$ is terminating, so for every term $s \in \mathcal{L}_{\trianglerighteq}(\mathcal{A}^k)$ 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:

- The initial automaton $\mathcal{A}$ must be $\mathcal{K}^n$-coherent and REFD.
- $\mathcal{R}$ must be terminating.
- 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 can 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 terms). Our approach looks for contracting equations for the set of closed 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* [25], 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 8. Assume that $\mathcal{C} = \{0 : 0, s : 1\}$.. For $k = 1$, the covering set is $\{s(x), 0\}$ and $\mathbb{E}^1_c = \{\{s(x) = x\}\}$. For depth 2, the 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 $\mathbb{E}^1_c$ and $\mathbb{E}^2_c$ satisfy Definition 3 and lead to different approximations.

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 algorithm searches for a set of contracting equations E_c such that verification succeeds, *i.e.* $\mathcal{L}(\mathcal{A}^*)$ 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 $\mathcal{A}_k^*$ recognising exactly the set $\mathcal{R}^*(L(\mathcal{A}_k))$ [20].
2. If $\mathcal{L}(\mathcal{A}_k^*)$ 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. Then for all remaining E_c, we 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 $\mathcal{A}^*$ 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 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 [28,26] and have corresponding TRSs in the $\mathcal{K}$ set defined above. The property φ consists in checking that a finite set of forbidden terms is not reachable (`Patterns` section of Timbuk specifications).

Given $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{A}^*$, the *correctness of the verification*, i.e. the fact that $\mathcal{L}(\mathcal{A}^*) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, can be checked in a proof assistant embedding a formalisation of rewriting and tree automata. It is enough to prove that (a) $\mathcal{L}(\mathcal{A}^*) \supseteq \mathcal{L}(\mathcal{A})$ and that (b) for all critical pairs $\langle l \to r, \sigma, q \rangle$ of $\mathcal{A}^*$ we have $r\sigma \to_{\mathcal{A}^*}^* q$. Property (a) can be checked using standard algorithms on tree automata. Property (b) can be checked by enumerating all critical pairs of $\mathcal{A}^*$ (there are finitely many) and by proving that all of them satisfy $r\sigma \to_{\mathcal{A}^*}^* q$. Since there exists algorithms for checking properties (a) and (b), the complete proof of correctness can automatically be built in the proof assistant. For instance, the automaton $\mathcal{A}^*$ can be used as a certificate to build the correctness proof in Coq [6] and in Isabelle/HOL [14]. It is also used to build unreachability proofs in Isabelle/HOL [14]. Besides, since verifying (a) and (b) is automatic, the correctness proof may be run outside of the proof assistant (in a more efficient way) using a formally verified external checker extracted from the formalisation. All our (successful) completion attempts output a `comp.res` file, containing $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{A}^*$, which has been certified automatically using the external certified checker of [6]. 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 [19].

Liquid Types [31], followed by Bounded Refinement Types [34,33], 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 be 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 [21] and Jones and Andersen [22]. 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 odd*). 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 [28] (PMRS). This technique improves the precision but is still not able to verify functions such as Example 1 (see [32] page 85). As shown in our experiments, our technique handles this example.

Kobayashi *et al.* developed a tree automata-based technique [26] (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 [26] 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 and that they can be used as certificates to build unreachability proofs in proof assistants (see Section 6).

Our verification framework is based on regular abstractions and uses a simple abstraction mechanism based on equations. Regular abstractions are less expressive than Higher-Order Recursion Schemes [29,23] or Collapsible Pushdown Au-

tomata [7], and equation-based abstractions are a particular case of predicate abstraction [24]. However, the two restrictions imposed in this particular framework result in two strong benefits. First, the precision of the approximation is formally defined and precisely controlled using equations: $\mathcal{L}(\mathcal{A}^*) \subseteq (\mathcal{R}/E)^*(\mathcal{L}(\mathcal{A}))$ [20]. This precision property permits us to prove intricate properties with simple (regular) abstractions. Second, using tree automata-based models facilitates the certification of the verification results in a proof assistant. This significantly increases the confidence in the verification result compared *e.g.*, to verdicts obtained by complex CEGAR-based model-checkers.

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 subset 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 question is whether this approach is complete: if there exists a regular approximation of the reachable terms of a functional program, can we build it using equations? We can already answered this question in the positive when $\mathcal{L} = \mathcal{W}(\mathcal{C})$, i.e., all results are first order terms [15]. Extending this result to all kind of results, including higher-order ones, 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 will 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 us to take a particular evaluation strategy into account, and in certain cases, improve the precision of the verification. We already experimented with this in [18]. 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 [24]. 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., Haudebourg, T., Jensen, T.: Verifying higher-order functional programs with tree automata : Extended version. https://hal.inria.fr/hal-01614380 (2017)
18. Genet, T., Salmon, Y.: Reachability Analysis of Innermost Rewriting – extended version. Logical Methods in Computer Science **13**(1) (2017) 1–35
19. Genet, T.: Termination criteria for tree automata completion. Journal of Logical and Algebraic Methods in Programming **85**(1) (2016) 3–33

20. Genet, T., Rusu, V.: Equational approximations for tree automata completion. Journal of Symbolic Computation **45**(5) (2010) 574–597
21. 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
22. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theoretical Computer Science **375**(1-3) (2007) 120–136
23. 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
24. 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
25. Kounalis, E.: Testing for the ground (co-)reducibility property in term-rewriting systems. Theor. Comput. Sci. **106**(1) (1992) 87–117
26. 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
27. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. TCS **403**(2-3) (2008) 239–264
28. 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
29. 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
30. Paulson, L.C., et al.: The isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory (1993)
31. 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
32. Salmon, Y.: Analyse d'atteignabilité pour les programmes fonctionnels avec stratégie d'évaluation en profondeur. Phd thesis, University of Rennes 1 (2015)
33. 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
34. 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