# Reachability Analysis of Innermost Rewriting

**Thomas Genet and Yann Salmon**

**IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France**
{Thomas.Genet,Yann.Salmon}@irisa.fr

## Abstract

We consider the problem of inferring a grammar describing the output of a functional program given a grammar describing its input. Solutions to this problem are helpful for detecting bugs or proving safety properties of functional programs and, several rewriting tools exist for solving this problem. However, known grammar inference techniques are not able to take evaluation strategies of the program into account. This yields very imprecise results when the evaluation strategy matters. In this work, we adapt the Tree Automata Completion algorithm to approximate accurately the set of terms reachable by rewriting under the innermost strategy. We prove that the proposed technique is sound and precise w.r.t. innermost rewriting. The proposed algorithm has been implemented in the Timbuk reachability tool. Experiments show that it noticeably improves the accuracy of static analysis for functional programs using the call-by-value evaluation strategy.

**1998 ACM Subject Classification** I.2.3 Deduction and Theorem Proving, F.4.2 Grammars and Other Rewriting Systems, D.2.4 Software/Program Verification

**Keywords and phrases** term rewriting systems, strategy, innermost strategy, tree automata, functional program, static analysis

**Digital Object Identifier** 10.4230/LIPIcs.RTA.2015.177

## 1 Introduction and motivations

If we define by a grammar the set of inputs of a functional program, is it possible to infer the grammar of its output? Some strongly typed functional programming languages (like Haskell, OCaml, Scala and F#) have a type inference mechanism. This mechanism, among others, permits to automatically detect some kinds of errors in the programs. In particular, when the inferred type is not the expected one, this suggests that there may be a bug in the function. To prove properties stronger than well typing of a program, it is possible to define properties and, then, to prove them using a proof assistant or an automatic theorem prover. However, defining those properties with logic formulas (and do the proof) generally requires a strong expertise.

Here, we focus on a restricted family of properties: regular properties on the structures manipulated by those programs. Using a grammar, we define the set of data structures given as input to a function and we want to infer the grammar that can be obtained as output (or an approximation). Like in the case of type inference, the output grammar can suggest that the program contains a bug, or on the opposite, that it satisfies a regular property.

The family of properties that can be shown in this way is restricted, but it strictly generalises standard typing as used in languages of the ML family[1]. There are other approaches where the type system is enriched by logic formulas and arithmetic like [29, 5], but they

[1] Standard types can easily be expressed as grammars. The opposite is not true. For instance, with a grammar one can distinguish between an empty and a non empty list.

26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 177–193
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

generally require to annotate the output of the function for type checking to succeed. The properties we consider here are intentionally simpler so as to limit as much as possible the need for annotations. The objective is to define a *lightweight* formal verification technique. The verification is *formal* because it *proves* that the results have a particular form. But, the verification is *lightweight* for two reasons. First, the proof is carried out automatically: no interaction with a prover or a proof assistant is necessary. Second, it is not necessary to state the property on the output of the function using complex logic formulas or an enriched type system but, instead, only to observe and check the result of an abstract computation.

With regards to the grammar inference technique itself, many works are devoted to this topic in the functional programming community [20, 25][2] as well as in the rewriting community [10, 27, 3, 14, 23, 2, 12]. In [12], starting from a term rewriting system (TRS for short) encoding a function and a tree automaton recognising the inputs of a function, it is possible to automatically produce a tree automaton *over-approximating* as precisely as possible the outputs. Note that a similar reasoning can be done on higher-order programs [16] using a well-known encoding of higher order functions into first-order TRS [20]. However, for the sake of simplicity, examples used in this paper will only be first order functions. This is implemented in the Timbuk tool [13]. Thus, we are close to building an *abstract interpreter*, evaluating a function on an (unbounded) regular set of inputs, for a real programming language. However, none of the aforementioned grammar inference techniques takes the evaluation strategy into account, though every functional programming language has one. As a consequence, those techniques produce very poor results as soon as the evaluation strategy matters or, as we will see, as soon as the program is not terminating. This paper proposes a grammar inference technique for the innermost strategy:

- overcoming the precision problems of [20, 25] and [10, 27, 3, 14, 23, 2, 12] on the analysis of functional programs using call-by-value strategy
- whose accuracy is not only shown on a practical point of view but also formally proved. This is another improvement w.r.t. other grammar inference techniques (except [14]).

## 1.1 Towards an abstract OCaml interpreter

In the following, we assume that we have an abstract OCaml interpreter. This interpreter takes a regular expression as an input and outputs another regular expression. In fact, all the computations presented in this way have been performed with Timbuk (and latter with TimbukSTRAT), but on a TRS and a tree automaton rather than on an OCaml function and a regular expression. We made this choice to ease the understanding of input and output languages, since regular expressions are far more easier to read and to understand than tree automata. Assume that we have a notation, inspired by regular expressions, to define regular languages of lists. Let us denote by `[a*]` (resp. `[a+]`) the language of lists having 0 (resp. 1) or more occurrences of symbol `a`. We denote by `[(a|b)*]` any list with 0 or more occurrences of `a` and `b` (in any order). Now, in OCaml, we define a function deleting all the occurrences of an element in a list. Here is a first (bugged) version of this function:

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then t else h::(delete x t);;
```

Of course, one can perform tests on this function using the usual OCaml interpreter:

[2] Note that the objective of other papers like [4, 21] is different. They aim at predicting the control flow of a program rather than estimating the possible results of a function (data flow).

```
# delete 2 [1;2;3]);;
-:int list= [1; 3]
```

With an *abstract* OCaml interpreter dealing with grammars, we could ask the following question: what is the set of the results obtained by applying `delete` to `a` and to any list of `a` and `b`?

```
# delete a [(a|b)*];;
-:abst list= [(a|b)*]
```

The obtained result is not the expected one. Since all occurrences of `a` should have been removed, we expected the result `[b*]`. Since the abstract interpreter results into a grammar *over-approximating* the set of outputs, this does not *show* that there is a bug, it only suggests it (like for type inference). Indeed, in the definition of `delete` there is a missing recursive call in the `then` branch. If we correct this mistake, we get:

```
# delete a [(a|b)*];;
-:abst list= [b*]
```

This result proves that `delete` deletes all occurrences of an element in a list. This is only one of the expected properties of `delete`, but shown automatically and without complex formalisation. Here is, in Timbuk syntax, the TRS $R$ and tree automata that are given to Timbuk to achieve the above proof.

```
Ops delete:2 cons:2 nil:0 a:0 b:0 ite:3 true:0 false:0 eq:2
Vars X Y Z
TRS R
eq(a,a)->true        eq(a,b)->false     eq(b,a)->false     eq(b,b)->true
delete(X,nil)->nil   ite(true,X,Y)->X   ite(false,X,Y)->Y
delete(X,cons(Y,Z))->ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
Automaton A0  States qf qa qb qlb qlab qnil  Final States qf
Transitions  delete(qa,qlab)->qf  a->qa  b->qb  nil->qlab
cons(qa,qlab)->qlab    cons(qb,qlab)->qlab
```

The resulting automaton computed by Timbuk is the following. It is not minimal but its recognised language is equivalent to `[b*]`.

```
States q0 q6 q8 Final States q6
Transitions cons(q8,q0)->q0  nil->q0  b->q8  cons(q8,q0)->q6  nil->q6
```

## 1.2 What is the problem with evaluation strategies?

Let us consider the function `sum(x)` which computes the sum of the `x` first natural numbers.

```
let rec sumList x y=                  let rec nth i (x::l)=
  (x+y)::(sumList (x+y) (y+1))           if i<=0 then x else nth (i-1) l
let sum x= nth x (sumList 0 0)
```

This function is terminating with call-by-need (used in Haskell) but not with call-by-value strategy (used in OCaml). Hence, any call to `sum` for any number `i` will not terminate because of OCaml's evaluation strategy. Thus the result of the abstract interpreter on `sum s*(0)` (*i.e.* `sum` applied to any natural number `0`, `s(0)`, ...) should be an empty grammar meaning that there is an empty set of results. However, if we use any of the techniques mentioned in the introduction to infer the output grammar, it will fail to show this. All those techniques compute reachable term grammars that do not take evaluation strategy into account. In particular, the inferred grammars will also contain all call-by-need evaluations. Thus, an abstract interpreter built on those techniques will produce a result of the form `s*(0)`, which is a very rough approximation. In this paper, we propose to improve the accuracy of such approximations by defining a language inference technique taking the call-by-value evaluation strategy into account.

### 1.3 Computing over-approximations of innermost reachable terms

Call-by-value evaluation strategy of functional programs is strongly related to innermost rewriting. The problem we are interested in is thus to compute (or to over-approximate) the set of innermost reachable terms. For a TRS $R$ and a set of terms $L_0 \subseteq T(\Sigma)$, the set of reachable terms is $R^*(L_0) = \{t \in T(\Sigma) \mid \exists s \in L_0, s \to_R^* t\}$. This set can be computed for specific classes of $R$ but, in general, it has to be approximated. Most of the techniques compute such approximations using tree automata (and not grammars) as the core formalism to represent or approximate the (possibly) infinite set of terms $R^*(L_0)$. Most of them also rely on a Knuth-Bendix completion-like algorithm to produce an automaton $\mathcal{A}^*$ recognising exactly, or over-approximating, the set of reachable terms. As a result, these techniques can be referred to as *tree automata completion* techniques [10, 27, 3, 14, 23].

Surprisingly, very little effort has been paid to computing or over-approximating the set $R^*_{strat}(L_0)$, *i.e.* set of reachable terms when $R$ is applied with a strategy *strat*. To the best of our knowledge, Pierre Réty and Julie Vuotto's work [26] is the first one to have tackled this goal. They give some sufficient conditions on $L_0$ and $R$ for $R^*_{strat}(L_0)$ to be recognised by a tree automaton $\mathcal{A}^*$, where *strat* can be the innermost or the outermost strategy. Innermost reachability for shallow TRSs was studied in [9]. However, in both cases, the restrictions on $R$ are strong and generally incompatible with functional programs seen as TRS. Moreover, the proposed techniques are not able to over-approximate reachable terms when the TRSs does not satisfy the restrictions.

In this paper, we concentrate on the innermost strategy and define a tree automata completion algorithm over-approximating the set $R^*_{\text{in}}(L_0)$ (innermost reachable terms) for any left-linear TRS $R$ and any regular set of input terms $L_0$. As the completion algorithm of [14], it is parameterized by a set of term equations $E$ defining the precision of the approximation. We prove the soundness of the algorithm: for all set of equation $E$, if completion terminates then the resulting automaton $\mathcal{A}^*$ recognises an over-approximation of $R^*_{\text{in}}(L_0)$. Then, we prove a precision theorem: $\mathcal{A}^*$ recognises no more terms than terms reachable by innermost rewriting with $R$ modulo equations of $E$. Finally, we show on examples that the precision of innermost completion noticeably improves the accuracy of the static analysis of functional programs.

This paper is organised as follows. Section 2 recalls some basic notions about TRSs and tree automata.

Section 3 exposes innermost completion. Section 4 states and proves the soundness of this method. Section 5 states the precision theorem. Section 6 demonstrates how our new technique can effectively give more precise results on functional programs thanks to the tool TimbukSTRAT, an implementation of our method in the Timbuk reachability tool [13].

## 2 Preliminaries

We use the same basic definitions and notions as in [1] and [28] for TRS and as in [6] for tree automata.

For a set of functions $\Sigma$ and a set of variables $\mathcal{X}$, we denote signatures by $(\Sigma, \mathcal{X})$, $T(\Sigma, \mathcal{X})$ for the set of terms and $T(\Sigma)$ for the set of ground terms over $(\Sigma, \mathcal{X})$. Given a signature $\Sigma$ and $k \in \mathbb{N}$, the set of its function symbols of arity $k$ is denoted by $\Sigma_k$.

▶ **Definition 1** (Rewriting rule, term rewriting system). A rewriting rule over $(\Sigma, \mathcal{X})$ is a couple $(\ell, r) \in \mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$, denoted by $\ell \to r$, such that $\ell$ is not a variable and any variable appearing in $r$ also appears in $\ell$. A term rewriting system (TRS) over $(\Sigma, \mathcal{X})$ is a set of rewriting rules over $(\Sigma, \mathcal{X})$.

The set of normal forms of a rewriting system $R$ (*i.e.* terms that are not reducible by $R$) is $\textsc{Irr}(R)$. A term $t$ is linear when no variable appears twice in $t$; a TRS is left-linear if the lhs of each of its rules is linear.

▶ **Definition 2** (Set of reachable terms). Given a signature $(\Sigma, \mathcal{X})$, a TRS $R$ over it and a set of terms $L \subseteq \mathcal{T}(\Sigma)$, we denote $R(L) = \{t \in \mathcal{T}(\Sigma) \mid \exists s \in L, s \to_R t\}$ and $R^*(L) = \{t \in \mathcal{T}(\Sigma) \mid \exists s \in L, s \to_R^* t\}$.

## 2.1 Equations

▶ **Definition 3** (Equivalence relation, congruence). A binary relation is an equivalence relation if it is reflexive, symmetric and transitive.

An equivalence relation $\equiv$ over $\mathcal{T}(\Sigma)$ is a congruence if for all $k \in \mathbb{N}$, for all $f \in \Sigma_k$, for all $t_1, \ldots, t_k, s_1, \ldots, s_k \in \mathcal{T}(\Sigma)$ such that $\forall i = 1 \ldots k$, $t_i \equiv s_i$, we have $f(t_1, \ldots, t_k) \equiv f(s_1, \ldots, s_k)$.

▶ **Definition 4** (Equation, $\equiv_E$). An equation over $(\Sigma, \mathcal{X})$ is a pair of terms $(s, t) \in \mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$, denoted by $s = t$. A set $E$ of equations over $(\Sigma, \mathcal{X})$ induces a congruence $\equiv_E$ over $\mathcal{T}(\Sigma)$ which is the smallest congruence over $\mathcal{T}(\Sigma)$ such that for all $s = t \in E$ and for all substitution $\theta : \mathcal{X} \to \mathcal{T}(\Sigma)$, $s\theta \equiv_E t\theta$. The equivalence classes of $\equiv_E$ are denoted with $[\cdot]_E$.

▶ **Definition 5** (Rewriting modulo $E$). Given a TRS $R$ and a set of equations $E$ both over $(\Sigma, \mathcal{X})$, we define the $R$ modulo $E$ rewriting relation, $\to_{R/E}$, as follows. For any $u, v \in \mathcal{T}(\Sigma)$, $u \to_{R/E} v$ if and only if there exist $u', v' \in \mathcal{T}(\Sigma)$ such that $u' \equiv_E u$, $v' \equiv_E v$ and $u' \to_R v'$. We define $\to_{R/E}^*$ as the reflexive and transitive closure of $\to_{R/E}$ and $(R/E)(L)$ and $(R/E)^*(L)$ in the same way as $R(L)$ and $R^*(L)$ where $\to_{R/E}$ replaces $\to_R$.

## 2.2 Tree automata

▶ **Definition 6** (Tree automaton, delta-transition, epsilon-transition, new state). An automaton over $\Sigma$ is some $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ where $Q$ is a finite set of states (symbols of arity 0 such that $\Sigma \cap Q = \varnothing$), $Q_F$ is a subset of $Q$ whose elements are called final states and $\Delta$ a finite set of transitions. A delta-transition is of the form $f(q_1, \ldots, q_k) \rightarrowtail q'$ where $f \in \Sigma_k$ and $q_1, \ldots, q_k, q' \in Q$. An epsilon-transition is of the form $q \rightarrowtail q'$ where $q, q' \in Q$. A configuration of $\mathcal{A}$ is a term in $\mathcal{T}(\Sigma, Q)$.

A state $q \in Q$ that appears nowhere in $\Delta$ is called a new state. A configuration is elementary if each of its sub-configurations at depth 1 (if any) is a state.

▶ **Definition 7.** Let $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ be an automaton and let $c, c'$ be configurations of $\mathcal{A}$. We say that $\mathcal{A}$ recognises $c$ into $c'$ in one step, and denoted by $c \underset{\mathcal{A}}{\rightarrowtail} c'$ if there a transition $\tau \rightarrowtail \rho$ in $\mathcal{A}$ and a context $C$ over $\mathcal{T}(\Sigma, Q)$ such that $c = C[\tau]$ and $c' = C[\rho]$. We denote by $\overset{*}{\underset{\mathcal{A}}{\rightarrowtail}}$ the reflexive and transitive closure of $\underset{\mathcal{A}}{\rightarrowtail}$ and, for any $q \in Q$, $\mathscr{L}(\mathcal{A}, q) = \left\{ t \in \mathcal{T}(\Sigma) \;\middle|\; t \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q \right\}$. We extend this definition to subsets of $Q$ and denote it by $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{A}, Q_F)$. A sequence of configurations $c_1, \ldots, c_n$ such that $t \underset{\mathcal{A}}{\rightarrowtail} c_1 \underset{\mathcal{A}}{\rightarrowtail} \cdots \underset{\mathcal{A}}{\rightarrowtail} c_n \underset{\mathcal{A}}{\rightarrowtail} q$ is called a recognition path for $t$ (into $q$) in $\mathcal{A}$. When $q \underset{\mathcal{A}}{\rightarrowtail} q'$ and $q' \underset{\mathcal{A}}{\rightarrowtail} q$, this is denoted by $q \underset{\mathcal{A}}{\rightleftarrows} q'$. A state $q$ of $\mathcal{A}$ is accessible if $\mathscr{L}(\mathcal{A}, q) \neq \varnothing$. An automaton is accessible if all of its states are.

▶ **Example 8.** Let $\Sigma$ be defined with $\Sigma_0 = \{n, 0\}$, $\Sigma_1 = \{s, a, f\}$, $\Sigma_2 = \{c\}$ where 0 is meant to represent integer zero, $s$ the successor operation on integers, $a$ the predecessor ('antecessor') operation, $n$ the empty list, $c$ the constructor of lists of integers and $f$ is intended to be the function on lists that filters out integer zero. Let $R = \{f(n) \to n, f(c(s(X), Y)) \to c(s(X), f(Y)), f(c(a(X), Y)) \to c(a(X), f(Y)), f(c(0, Y)) \to f(Y), a(s(X)) \to X, s(a(X)) \to X\}$. Let $\mathcal{A}_0$ be the tree automaton with final state $q_f$ and transitions $\{n \rightarrowtail q_n, 0 \rightarrowtail q_0, s(q_0) \rightarrowtail q_s, a(q_s) \rightarrowtail q_a, c(q_a, q_n) \rightarrowtail q_c, f(q_c) \rightarrowtail q_f\}$. We have $\mathscr{L}(\mathcal{A}_0, q_f) = \{f(c(a(s(0)), n))\}$ and $R(\mathscr{L}(\mathcal{A}_0, q_f)) = \{f(c(0, n)), c(a(s(0)), f(n))\}$.

▶ Remark. Automata transitions may have 'colours', like $\mathfrak{R}$ for transition $q \overset{\mathfrak{R}}{\rightarrowtail} q'$. We will use colours $\mathfrak{R}$ and $\mathfrak{E}$ for transitions denoting either rewrite or equational steps.

▶ **Definition 9.** Given an automaton $\mathcal{A}$ and a colour $\mathfrak{R}$, we denote by $\mathcal{A}^{\cancel{\mathfrak{R}}}$ the automaton obtained from $\mathcal{A}$ by removing all transitions coloured with $\mathfrak{R}$.

## 2.3 Pair automaton

We now give notations used for pair automaton, the archetype of which is the product of two automata.

▶ **Definition 10** (Pair automaton). An automaton $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ is said to be a pair automaton if there exists some sets $Q_1$ and $Q_2$ such that $Q = Q_1 \times Q_2$.

▶ **Definition 11** (Product automaton [6]). Let $\mathcal{A} = (\Sigma, Q, Q_F, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (\Sigma, P, P_F, \Delta_{\mathcal{B}})$ be two automata. The product automaton of $\mathcal{A}$ and $\mathcal{B}$ is $\mathcal{A} \times \mathcal{B} = (\Sigma, Q \times P, Q_F \times P_F, \Delta)$ where $\Delta = \{f(\langle q_1, p_1\rangle, \ldots, \langle q_k, p_k\rangle) \rightarrowtail \langle q', p'\rangle \mid f(q_1, \ldots, q_k) \rightarrowtail q' \in \Delta_{\mathcal{A}} \wedge f(p_1, \ldots, p_k) \rightarrowtail p' \in \Delta_{\mathcal{B}}\} \cup \{\langle q, p\rangle \rightarrowtail \langle q', p\rangle \mid p \in P,\ q \rightarrowtail q' \in \Delta_{\mathcal{A}}\} \cup \{\langle q, p\rangle \rightarrowtail \langle q, p'\rangle \mid q \in Q,\ p \rightarrowtail p' \in \Delta_{\mathcal{B}}\}$

▶ **Definition 12** (Projections). Let $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ be a pair automaton, let $\tau \rightarrowtail \rho$ be one of its transitions and $\langle q, p\rangle$ be one of its states. We define $\Pi_1(\langle q, p\rangle) = q$ and extend $\Pi_1(\cdot)$ to configurations inductively: $\Pi_1(f(\gamma_1, \ldots, \gamma_k)) = f(\Pi_1(\gamma_1), \ldots, \Pi_1(\gamma_k))$. We define $\Pi_1(\tau \rightarrowtail \rho) = \Pi_1(\tau) \rightarrowtail \Pi_1(\rho)$. We define $\Pi_1(\mathcal{A}) = (\Sigma, \Pi_1(Q), \Pi_1(Q_F), \Pi_1(\Delta))$. $\Pi_2(\cdot)$ is defined on all these objects in the same way for the right component.

▶ Remark. Using $\Pi_1(\mathcal{A})$ amounts to forgetting the precision given by the right component of the states. As a result, $\mathscr{L}(\Pi_1(\mathcal{A}), q) \supseteq \bigcup\limits_{p \in P} \mathscr{L}(\mathcal{A}, \langle q, p\rangle)$.

## 2.4 Innermost strategy

In general, a strategy over a TRS $R$ is a set of (computable) criteria to describe a certain sub-relation of $\to_R$. In this paper, we will be interested in innermost strategies. In these strategies, commonly used to execute functional programs ('call-by-value'), terms are rewritten by always contracting one of the lowest reducible subterms. If $s \to_R t$ and rewriting occurs at a position $p$ of $s$, $s|_p$ is called the *redex*.

▶ **Definition 13** (Innermost strategy). Given a TRS $R$ and two terms $s, t$, we say that $s$ can be rewritten into $t$ by $R$ with an innermost strategy, denoted by $s \to_{R_{\text{in}}} t$, if $s \to_R t$ and each strict subterm of the redex in $s$ is a $R$-normal form. We define $R_{\text{in}}(L)$ and $R^*_{\text{in}}(L)$ in the same way as $R(L)$, $R^*(L)$ where $\to_{R_{\text{in}}}$ replaces $\to_R$.

▶ **Example 14.** We continue on Example 8. We have $R_{\text{in}}(\mathscr{L}(\mathcal{A}_0, q_f)) = \{f(c(0, n))\}$ because the rewriting step $f(c(a(s(0)), n)) \to_R c(a(s(0)), f(n))$ is not innermost since the subterm $a(s(0))$ of the redex $f(c(a(s(0)), n))$ is not in normal form.

To deal with innermost strategies, we have to discriminate normal forms. When $R$ is left-linear, it is possible to compute a tree automaton recognising normal forms.

▶ **Theorem 15** ([7])**.** *Let $R$ be a left-linear TRS. There is a deterministic and complete tree automaton $\mathcal{A}\mathcal{IRR}(R)$ whose states are all final except one, denoted by $p_{\text{red}}$ and such that $\mathscr{L}(\mathcal{A}\mathcal{IRR}(R)) = \textsc{Irr}(R)$ and $\mathscr{L}(\mathcal{A}\mathcal{IRR}(R), p_{\text{red}}) = T(\Sigma) \smallsetminus \textsc{Irr}(R)$.*

▶ Remark. Since $\mathcal{A}\mathcal{IRR}(R)$ is deterministic, for any state $p \neq p_{\text{red}}$, $\mathscr{L}(\mathcal{A}\mathcal{IRR}(R), p) \subseteq \textsc{Irr}(R)$.

▶ Remark. If a term $s$ is reducible, any term having $s$ as a subterm is also reducible. Thus any transition of $\mathcal{A}\mathcal{IRR}(R)$ where $p_{\text{red}}$ appears in the left-hand side will necessarily have $p_{\text{red}}$ as its right-hand side. Thus, for brevity, these transitions will always be left implicit when describing the automaton $\mathcal{A}\mathcal{IRR}(R)$ for some TRS $R$.

▶ **Example 16.** In Example 8, $\mathcal{A}\mathcal{IRR}(R)$ needs, in addition to $p_{\text{red}}$, a state $p_{list}$ to recognise lists of integers, a state $p_a$ for terms of the form $a(\dots)$, a state $p_s$ for $s(\dots)$, a state $p_0$ for 0 and a state $p_{var}$ to recognise terms that are not subterms of lhs of $R$, but may participate in building a reducible term by being instances of variables in a lhs. We note $P = \{p_{list}, p_0, p_a, p_s, p_{var}\}$ and $P_{int} = \{p_0, p_a, p_s\}$. The interesting transitions are thus $0 \rightarrowtail p_0$, $\bigcup_{p \in P \smallsetminus \{p_a\}} \{s(p) \rightarrowtail p_s\}$, $\bigcup_{p \in P \smallsetminus \{p_s\}} \{a(p) \rightarrowtail p_a\}$ ; $n \rightarrowtail p_{list}$, $\bigcup_{p \in P_{int}, p' \in P} \{c(p, p') \rightarrowtail p_{list}\}$ ; $f(p_{list}) \rightarrowtail p_{\text{red}}$, $a(p_s) \rightarrowtail p_{\text{red}}$, $s(p_a) \rightarrowtail p_{\text{red}}$. Furthermore, as remarked above, any configuration that contains a $p_{\text{red}}$ is recognised into $p_{\text{red}}$. Finally, some configurations are not covered by the previous cases: they are recognised into $p_{var}$.

## 3 Innermost equational completion

Our first contribution is an adaptation of the classical equational completion of [14], which is an iterative process on automata. Starting from a tree automaton $\mathcal{A}_0$ it iteratively computes tree automata $\mathcal{A}_1, \mathcal{A}_2, \dots$ until a fixpoint automaton $\mathcal{A}_*$ is found. Each iteration comprises two parts: (exact) completion itself (Subsection 3.1), then equational merging (Subsection 3.2). The former tends to incorporate descendants by $R$ of already recognised terms into the recognised language; this leads to the creation of new states. The latter tends to merge states in order to ease termination of the overall process, at the cost of precision of the computed result. Some transition added by equational completion will have colours $\mathfrak{R}$ or $\mathfrak{E}$. We will use colours $\mathfrak{R}$ and $\mathfrak{E}$ for transitions denoting either rewrite or equational steps; it is assumed that the transitions of the input automaton $\mathcal{A}_0$ do not have any colour and that $\mathcal{A}_0$ does not have any epsilon-transition.

The equational completion of [14] is blind to strategies. To make it innermost-strategy-aware, we equip each state of the studied automata with a state from the automaton $\mathcal{A}\mathcal{IRR}(R)$ (see Theorem 15) to keep track of normal and reducible forms. Let $\mathcal{A}_{init}$ be an automaton recognising the initial language. Completion will start with $\mathcal{A}_0 = \mathcal{A}_{init} \times \mathcal{A}\mathcal{IRR}(R)$. This automaton enjoys the following property.

▶ **Definition 17** (Consistency with $\mathcal{A}\mathcal{IRR}(R)$)**.** A pair automaton $\mathcal{A}$ is said to be consistent with $\mathcal{A}\mathcal{IRR}(R)$ if, for any configuration $c$ and any state $\langle q, p \rangle$ of $\mathcal{A}$, $\Pi_2(c)$ is a configuration of $\mathcal{A}\mathcal{IRR}(R)$ and $p$ is a state of $\mathcal{A}\mathcal{IRR}(R)$, and if $c \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p \rangle$ then $\Pi_2(c) \overset{*}{\underset{\mathcal{A}\mathcal{IRR}(R)}{\rightarrowtail}} p$.

$$\begin{tikzcd}
\ell\sigma \arrow[r, "R"] \arrow[d, tail, "\mathcal{A}"', "*"] & r\sigma \arrow[dd, tail, dashed, "*"', "\mathcal{A}'"] \\
f(q_1,\ldots,q_k) \arrow[d, tail, "\mathcal{A}"'] & \\
q & q' \arrow[l, tail, dashed, "\mathfrak{R}"', "\mathcal{A}'"]
\end{tikzcd}$$

**(a)** For comparison: Classical completion.

$$\begin{tikzcd}
\ell\sigma \arrow[r, "R_{\text{in}}"] \arrow[d, tail, "\mathcal{A}"', "*"] & r\sigma \arrow[dd, tail, dashed, "*"', "\mathcal{A}'"] \\
f(\langle q_1,p_1\rangle,\ldots,\langle q_k,p_k\rangle) \arrow[d, tail, "\mathcal{A}"', "p_1,\ldots,p_k \neq p_{\text{red}}"] & \\
\langle q,p_{\text{red}}\rangle & \langle q',p_{r\sigma}\rangle \arrow[dl, tail, dashed, "\mathfrak{R}"', "\mathcal{A}'"] \\
\langle q,p_{r\sigma}\rangle &
\end{tikzcd}$$

**(b)** Innermost completion: we added information about normal forms.

**Figure 1** Comparison of classical and innermost critical pairs.

## 3.1 Exact completion

The first step of equational completion incorporates descendants by $R$ of terms recognised by $\mathcal{A}_i$ into $\mathcal{A}_{i+1}$. The principle is to search for critical pairs between $\mathcal{A}_i$ and $R$. In classical completion, a critical pair is triple $(\ell \to r, \sigma, q)$ such that $l\sigma \xrightarrow[\mathcal{A}_i]{*} q$, $l\sigma \to_R r\sigma$ and $r\sigma \not\xrightarrow[\mathcal{A}_i]{*} q$. Such a critical pair denotes a rewriting position of a term recognised by $\mathcal{A}_i$ such that the rewritten term is not recognised by $\mathcal{A}_i$. For the innermost strategy, the critical pair notion is slightly refined since it also needs that every subterm $\gamma$ at depth 1 in $\ell\sigma$ is in normal form. This corresponds to the third case of the following definition where $\gamma \xrightarrow[\mathcal{A}]{*} \langle q_\gamma, p_\gamma\rangle$ and $p_\gamma \neq p_{red}$ ensures that $\gamma$ is irreducible. See Figure 1.

▶ **Definition 18** (Innermost critical pair)**.** Let $\mathcal{A}$ be a pair automaton. A tuple $(\ell \to r, \sigma, \langle q, p\rangle)$ where $\ell \to r \in R$, $\sigma : \mathcal{X} \to Q_{\mathcal{A}}$ and $\langle q, p\rangle \in Q_{\mathcal{A}}$ is called a critical pair if

1. $\ell\sigma \xrightarrow[\mathcal{A}]{*} \langle q, p\rangle$,
2. there is no $p'$ such that $r\sigma \xrightarrow[\mathcal{A}]{*} \langle q, p'\rangle$ and
3. for each sub-configuration $\gamma$ at depth 1 of $\ell\sigma$, the state $\langle q_\gamma, p_\gamma\rangle$ such that $\gamma \xrightarrow[\mathcal{A}]{*} \langle q_\gamma, p_\gamma\rangle$ in the recognition path of condition 1 is with $p_\gamma \neq p_{\text{red}}$.

▶ Remark. Because a critical pair denotes a rewriting situation, the $p$ of Definition 18 is necessarily $p_{\text{red}}$ as long as $\mathcal{A}$ is consistent with $\mathcal{A}_{\mathcal{IRR}}(R)$.

▶ **Example 19.** In the situation of Examples 8 and 16, consider the rule $f(c(a(X), Y)) \to c(a(X), f(Y))$, the substitution $\sigma_1 = \{X \mapsto \langle q_s, p_s\rangle, Y \mapsto \langle q_n, p_n\rangle\}$ and the state $\langle q_f, p_{\text{red}}\rangle$: this is not an innermost critical pair because the recognition path is:

$f(c(a(\langle q_s, p_s\rangle), \langle q_n, p_n\rangle)) \rightarrowtail f(c(\langle q_a, p_{\text{red}}\rangle, \langle q_n, p_n\rangle)) \rightarrowtail f(\langle q_c, p_{\text{red}}\rangle) \rightarrowtail \langle q_f, p_{\text{red}}\rangle$

and so there is a $p_{\text{red}}$ at depth 1. But there is an innermost critical pair in $\mathcal{A}_0$ with the rule $a(s(X)) \to X$, the substitution $\sigma_2 = \{X \mapsto \langle q_0, p_0\rangle\}$ and the state $\langle q_a, p_{\text{red}}\rangle$.

Once a critical pair is found, the completion algorithm needs to resolve it: it adds the necessary transitions for $r\sigma$ to be recognised by the completed automaton. Classical completion adds the necessary transitions so that $r\sigma \xrightarrow[\mathcal{A}']{*} q$, where $\mathcal{A}'$ is the completed automaton. In innermost completion this is more complex. The state $q$ is, in fact, a pair of the form $\langle q, p_{red}\rangle$ and adding transitions so that $r\sigma \xrightarrow[\mathcal{A}']{*} \langle q, p_{red}\rangle$ may jeopardise consistency of $\mathcal{A}'$ with $\mathcal{A}_{\mathcal{IRR}}$ if $r\sigma$ is not reducible. Thus the diagram is closed in a different way preserving consistency with $\mathcal{A}_{\mathcal{IRR}}$ (see Figure 1). However, like in classical completion, this can generally not be done in one step, as $r\sigma$ might be a non-elementary configuration. We have to split the configuration into elementary configurations and to introduce new states to recognise them: this is what *normalisation* (denoted by $\mathit{Norm}_{\mathcal{A}}$) does. Given an automaton $\mathcal{A}$, a configuration $c$ of $\mathcal{A}$ and a new state $\langle q, p\rangle$, we denote by $\mathit{Norm}_{\mathcal{A}}(c, \langle q, p\rangle)$ the set of transitions (with new states) that we add to $\mathcal{A}$ to ensure that $c$ is recognised into $\langle q, p\rangle$. The $\mathit{Norm}_{\mathcal{A}}$ operation is parameterized by $\mathcal{A}$ because it reuses transitions of $\mathcal{A}$ whenever it is possible. On an example, we show how normalisation behaves. For a formal definition see [15].

▶ **Example 20.** With a suitable signature, suppose that automaton $\mathcal{A}$ consists of the transitions $c \rightarrowtail \langle q_1, p_c\rangle$ and $f(\langle q_1, p_c\rangle) \rightarrowtail \left\langle q_2, p_{f(c)}\right\rangle$ and we want to normalise $f(g(\left\langle q_2, p_{f(c)}\right\rangle, c))$ to the new state $\left\langle q_N, p_{f(g(f(c),c))}\right\rangle$. We first have to normalise under $g$: $\left\langle q_2, p_{f(c)}\right\rangle$ is already a state, so it does not need to be normalised; $c$ has to be normalised to a state: since $\mathcal{A}$ already has transition $c \rightarrowtail \langle q_1, p_c\rangle$, we add no new state and it remains to normalise $g(\left\langle q_2, p_{f(c)}\right\rangle, \langle q_1, p_c\rangle)$. Since $\mathcal{A}$ does not contain a transition for this configuration, we must add a new state $\left\langle q', p_{g(f(c),c)}\right\rangle$ and the transition $g(\left\langle q_2, p_{f(c)}\right\rangle, \langle q_1, p_c\rangle) \rightarrowtail \left\langle q', p_{g(f(c),c)}\right\rangle$. Finally, we add $f(\left\langle q', p_{g(f(c),c)}\right\rangle) \rightarrowtail \left\langle q_N, p_{f(g(f(c),c))}\right\rangle$. Note that due to consistency with $\mathcal{AIRR}(R)$, whenever we add a new transition $c' \rightarrowtail \langle q', p'\rangle$, only the $q'$ is arbitrary: the $p'$ is always the state of $\mathcal{AIRR}(R)$ such that $\Pi_2\,(c) \underset{\mathcal{AIRR}(R)}{\rightarrowtail} p'$, in order to preserve consistency with $\mathcal{AIRR}(R)$.

Completion of a critical pair is done in two steps. The first set of operations formalises 'closing the square' (see Figure 1), *i.e.* if $l\sigma \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p_{red}\rangle$ then we add transitions $r\sigma \overset{*}{\underset{\mathcal{A}'}{\rightarrowtail}} \langle q', p_{r\sigma}\rangle \overset{\mathfrak{R}}{\rightarrowtail} \langle q, p_{r\sigma}\rangle$. The second step adds the necessary transitions for any context $C[r\sigma]$ to be recognised in the tree automaton if $C[l\sigma]$ was. Thus if the the recognition path for $C[l\sigma]$ is of the form $C[l\sigma] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\langle q, p_{red}\rangle] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_c, p_{red}\rangle$, we add the necessary transitions for $C[\langle q, p_{r\sigma}\rangle]$ to be recognised into $\langle q_c, p_c\rangle$ where $p_c$ is the state of $\mathcal{AIRR}(R)$ recognising $C[r\sigma]$.

▶ **Definition 21** (Completion of an innermost critical pair)**.** A critical pair $(\ell \to r, \sigma, \langle q, p\rangle)$ in automaton $\mathcal{A}$ is completed by first computing $N = \mathit{Norm}_{\mathcal{A}^{\not\mathfrak{R}}}(r\sigma, \langle q', p_{r\sigma}\rangle)$ where $q'$ is a new state and $\Pi_2\,(r\sigma) \overset{*}{\underset{\mathcal{AIRR}(R)}{\rightarrowtail}} p_{r\sigma}$, then adding to $\mathcal{A}$ the new states and the transitions appearing in $N$ as well as the transition $\langle q', p_{r\sigma}\rangle \overset{\mathfrak{R}}{\rightarrowtail} \langle q, p_{r\sigma}\rangle$. If $r\sigma$ is a trivial configuration (*i.e.* $r$ is just a variable, and thus $\Pi_2\,(r\sigma)$ is a state), only transition $r\sigma \overset{\mathfrak{R}}{\rightarrowtail} \langle q, \Pi_2\,(r\sigma)\rangle$ is added. Afterwards, we execute the following supplementary operations. For any new transition $f(\ldots, \langle q, p_{\mathrm{red}}\rangle, \ldots) \rightarrowtail \langle q'', p''\rangle$, we add a transition $f(\ldots, \langle q, p_{r\sigma}\rangle, \ldots) \rightarrowtail \langle q'', p'''\rangle$ with $f(\ldots, p_{r\sigma}, \ldots) \underset{\mathcal{AIRR}(R)}{\rightarrowtail} p'''$. These new transitions are in turn recursively considered for the supplementary operations[3].

▶ **Definition 22** (Innermost completion step)**.** Let $PC$ be the set of all innermost critical pairs of $\mathcal{A}_i$. For $pc \in PC$, let $N_{pc}$ be the set of new states and transitions needed under Definition 21 to complete $pc$, and $\mathcal{A} \cup N_{pc}$ the automaton $\mathcal{A}$ completed by states and transitions of $N_{pc}$. Then $\mathcal{A}_{i+1} = \mathcal{A}_i \cup \bigcup\limits_{pc \in PC} N_{pc}$.

[3] Those supplementary operations add new pairs, but the element of each pair are not new. So, this necessarily terminates.

▶ **Lemma 23.** *Let $\mathcal{A}$ be an automaton obtained from some $\mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$ after some steps of innermost completion. $\mathcal{A}$ is consistent with $\mathcal{A}_{\mathcal{IRR}}(R)$.*

Due to space constraints, the full proofs can be found in [15].

## 3.2 Equational simplification

▶ **Definition 24.** Given two states $q$, $q'$ of some automaton $\mathcal{A}$ and a colour $\mathfrak{C}$, we note $q \overset{\mathfrak{C}}{\underset{\mathcal{A}}{\rightleftarrows}} q'$ when we have both $q \overset{\mathfrak{C}}{\underset{\mathcal{A}}{\rightarrowtail}} q'$ and $q' \overset{\mathfrak{C}}{\underset{\mathcal{A}}{\rightarrowtail}} q$.

▶ **Definition 25** (Situation of application of an equation)**.** Given an equation $s = t$, an automaton $\mathcal{A}$, a substitution $\theta : \mathcal{X} \to Q_{\mathcal{A}}$ and states $\langle q_1, p_1 \rangle$ and $\langle q_2, p_2 \rangle$, we say that $(s = t, \theta, \langle q_1, p_1 \rangle, \langle q_2, p_2 \rangle)$ is a situation of application in $\mathcal{A}$ if

1. $s\theta \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_1, p_1 \rangle$,
2. $t\theta \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_2, p_2 \rangle$,
3. $\langle q_1, p_1 \rangle \overset{\mathfrak{C}}{\underset{\mathcal{A}}{\not\rightleftarrows}} \langle q_2, p_2 \rangle$
4. $p_1 = p_2$.

Note that when $p_1 \neq p_2$, this is not a situation of application for an equation. This is the only difference with the situation of application in classical completion. This restriction avoids, in particular, to apply an equation between reducible and irreducible terms. Such terms will be recognised by states having two distinct second components. On the opposite, when a situation of application arises, we 'apply' the equation, *i.e.* add the necessary transitions to have $\langle q_1, p_1 \rangle \overset{\mathfrak{C}}{\underset{\mathcal{A}}{\rightleftarrows}} \langle q_2, p_2 \rangle$ and supplementary transitions to lift this property to any embedding context. We apply equations until there are no more situation of application on the automaton (this is guaranteed to happen because we add no new state in this part).

▶ **Definition 26** (Application of an equation)**.** Given $(s = t, \theta, \langle q_1, p_1 \rangle, \langle q_2, p_1 \rangle)$ a situation of application in $\mathcal{A}$, applying the underlying equation in it consists in adding transitions $\langle q_1, p_1 \rangle \overset{\mathfrak{C}}{\rightarrowtail} \langle q_2, p_1 \rangle$ and $\langle q_2, p_1 \rangle \overset{\mathfrak{C}}{\rightarrowtail} \langle q_1, p_1 \rangle$ to $\mathcal{A}$. We also add the supplementary transitions $\langle q_1, p_1' \rangle \overset{\mathfrak{C}}{\rightarrowtail} \langle q_2, p_1' \rangle$ and $\langle q_2, p_1' \rangle \overset{\mathfrak{C}}{\rightarrowtail} \langle q_1, p_1' \rangle$ where $\langle q_1, p_1' \rangle$ and $\langle q_2, p_1' \rangle$ occur in the automaton.

▶ **Lemma 27.** *Applying an equation preserves consistency with $\mathcal{A}_{\mathcal{IRR}}(R)$.*

## 3.3 Innermost completion and equations

▶ **Definition 28** (Step of innermost equational completion)**.** Let $R$ be a left-linear TRS, $\mathcal{A}_{init}$ a tree automaton, $E$ a set of equations and $\mathcal{A}_0 = \mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$. The automaton $\mathcal{A}_{i+1}$ is obtained, from $\mathcal{A}_i$, by applying an innermost completion step on $\mathcal{A}_i$ (Definition 21) and solving all situations of applications of equations of $E$ (Definition 25).

# 4 Correctness

▶ **Definition 29** (Correct automaton)**.** An automaton $\mathcal{A}$ is correct w.r.t. $R_{\text{in}}$ if for all states $\langle q, p_{\text{red}} \rangle$ of $\mathcal{A}$, for all $u \in \mathscr{L}(\mathcal{A}, \langle q, p_{\text{red}} \rangle)$ and for all $v \in R_{\text{in}}(u)$, either there is a state $p$ of $\mathcal{A}_{\mathcal{IRR}}(R)$ such that $v \in \mathscr{L}(\mathcal{A}, \langle q, p \rangle)$ or there is a critical pair $(\ell \to r, \sigma, \langle q_0, p_0 \rangle)$ in $\mathcal{A}$ for some $\langle q_0, p_0 \rangle$ and a context $C$ on $T(\Sigma)$ such that $u \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\ell\sigma] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\langle q_0, p_{\text{red}} \rangle] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p_{\text{red}} \rangle$ and $v \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[r\sigma]$.

▶ **Lemma 30.** *Any automaton produced by innermost equational completion starting from some* $\mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$ *is correct w.r.t.* $R_{\text{in}}$.

▶ **Theorem 31** (Correctness). *Assuming* $R$ *is left-linear, the innermost equational completion procedure defined above produces a correct result whenever it terminates and produces some fixpoint* $\mathcal{A}_{\text{in}*}$*:*

$$\mathscr{L}(\mathcal{A}_{\text{in}*}) \supseteq R_{\text{in}}^*(\mathscr{L}(\mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R))).$$

**Proof.** $\mathcal{A}_{\text{in}*}$ is correct w.r.t. $R_{\text{in}}$, but the case of Definition 29 where there remains a critical pair cannot occur, because it is a fixpoint. ◀

## 5 Precision theorem

We just showed that the approximation is correct. Now we investigate its accuracy on a theoretical point of view. This theorem is technical and difficult to prove (details can be found in [15]). But, this result is crucial because producing an over-approximation of reachable terms is easy (the tree automaton recognising $T(\Sigma)$ is a correct over-approximation) but producing an accurate approximation is hard. To the best of our knowledge, no other work dealing with abstract interpretation of functional programs or computing approximations of regular languages can provide such a formal precision guarantee (except [14] but in the case of general rewriting). Like in [14], we formally quantify the accuracy w.r.t. rewriting modulo $E$, replaced here by *innermost* rewriting modulo $E$. The relation of innermost rewriting modulo $E$, denoted by $\rightarrow_{R_{\text{in}}/E}$, is defined as rewriting modulo $E$ where $\rightarrow_{R_{\text{in}}}$ replaces $\rightarrow_R$. We also define $(R_{\text{in}}/E)(L)$ and $(R_{\text{in}}/E)^*(L)$ in the same way as $R(L)$, $R^*(L)$ where $\rightarrow_{R_{\text{in}}/E}$ replaces $\rightarrow_R$.

The objective of the proof is to show that the completed tree automaton recognises no more terms than those reachable by $R_{\text{in}}/E$ rewriting. The accuracy relies on the $R_{\text{in}}/E$-coherence property of the completed tree automaton, defined below. Roughly, a tree automaton $\mathcal{A}$ is $R_{\text{in}}/E$-coherent if $\overset{*}{\underset{\mathcal{A}}{\rightarrowtail}}$ is coherent w.r.t. $R$ innermost rewriting steps and $E$ equational steps. More precisely if $s \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q$ and $t \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q$ with no epsilon transitions with colour $\mathfrak{R}$, then $s =_E t$ (this is called separation of $E$-classes for $\mathcal{A}^{\cancel{\mathfrak{R}}}$). And, if $t \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q$ with at least one epsilon transitions with colour $\mathfrak{R}$, then $s \rightarrow^*_{R_{\text{in}}/E} t$ (this is called $R_{\text{in}}$-coherence of $\mathcal{A}$). Roughly, a tree automaton separates $E$-classes if all terms recognized by a state are $E$-equivalent. Later, we will require this property on $\mathcal{A}_0$ and then propagate it on $\mathcal{A}_i^{\cancel{\mathfrak{R}}}$, for all completed automata $\mathcal{A}_i$.

▶ **Definition 32** (Separation of $E$-classes). The pair automaton $\mathcal{A}$ separates the classes of $E$ if for any $q \in \Pi_1(Q_\mathcal{A})$, there is a term $s$ such that for all $p \in \Pi_2(Q_\mathcal{A})$, $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \subseteq [s]_E$. We denote by $[q]_E^\mathcal{A}$ the common class of terms in $\mathscr{L}(\mathcal{A}, \langle q, \cdot\rangle)$, and extend this to configurations. We say the separation of classes by $\mathcal{A}$ is total if $\Pi_1(\mathcal{A})$ is accessible.

▶ **Definition 33** ($R_{\text{in}}/E$-coherence). An automaton $\mathcal{A}$ is $R_{\text{in}}/E$-coherent if

1. $\mathcal{A}^{\cancel{\mathfrak{R}}}$ totally separates the classes of $E$,
2. $\mathcal{A}$ is accessible, and
3. for any state $\langle q, p\rangle$ of $\mathcal{A}$, $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \subseteq (R_{\text{in}}/E)^* \left([q]_E^{\mathcal{A}^{\cancel{\mathfrak{R}}}}\right)$.

Then, the objective is to show that the two basic elements of innermost equational completion: completing a critical pair and applying an equation preserve $R_{\text{in}}/E$-coherence. This is the purpose of the two following lemmas.

▶ **Lemma 34.** *Completion of an innermost critical pair preserves $R_{\text{in}}/E$-coherence.*

▶ **Lemma 35.** *Equational simplification preserves $R_{\text{in}}/E$-coherence.*

This shows that, under the assumption that $\mathcal{A}_0$ separates the classes of $E$, innermost equational completion will never never add to the computed approximation a term that is not a descendant of $\mathscr{L}(\mathcal{A}_0)$ through $R_{\text{in}}$ modulo $E$ rewriting. This permits to state the main theorem, which formally defines the precision of the completed tree automaton.

▶ **Theorem 36** (Precision)**.** *Let $E$ be a set of equations. Let $\mathcal{A}_0 = \mathcal{A}_{init} \times \mathcal{A}\mathcal{IRR}(R)$, where $\mathcal{A}_{init}$ has designated final states. We prune $\mathcal{A}_0$ of its non-accessible states. Suppose $\mathcal{A}_0$ separates the classes of $E$. Let $R$ be any left-linear TRS. Let $\mathcal{A}_i$ be obtained from $\mathcal{A}_0$ after some steps of innermost equational completion. Then*

$$\mathscr{L}(\mathcal{A}_i) \subseteq (R_{\text{in}}/E)^*(\mathscr{L}(\mathcal{A}_0))).$$

**Proof.** (Sketch) We know that $\mathcal{A}_0$ is $R_{\text{in}}/E$-coherent because (1) $\mathcal{A}_0^{\cancel{\mathfrak{X}}}$ separates the classes of $E$ ($\mathcal{A}_0$ separates the classes of $E$ and $\mathcal{A}_0 = \mathcal{A}_0^{\cancel{\mathfrak{X}}}$ since none of $\mathcal{A}_{init}$ and $\mathcal{A}\mathcal{IRR}$ have epsilon transitions), and (2) $\mathcal{A}_0$ is accessible. Condition (3) of Definition 33 is trivially satisfied since $\mathcal{A}_0$ separates classes of $E$, meaning that for all states $q$, there is a term $s$ s.t. $\mathscr{L}(\mathcal{A}_0, \langle q, p\rangle) \subseteq [s]_E$, *i.e.* all terms recognized by $q$ are $E$-equivalent to $s$ which is a particular case of case (3) in Definition 33. Then, during successive completion steps, by Lemma 34 and 35, we know that each basic transformation applied on $\mathcal{A}_0$ (completion or equational step) will preserve the $R_{\text{in}}/E$-coherence of $\mathcal{A}_0$. Thus, $\mathcal{A}_i$ is $R_{\text{in}}/E$-coherent. Finally, case (3) of $R_{\text{in}}/E$-coherence of $\mathcal{A}_i$ entails the result. ◀

Note that the fact that $\mathcal{A}_0$ needs to separate the classes of $E$ is not a strong restriction. In the particular case of functional TRS (TRS encoding first order typed functional programs [12]), there always exists a tree automaton recognising a language equal to $\mathscr{L}(\mathcal{A}_0)$ and which separates the classes of $E$, see [11] for details.

## 6 Improving accuracy of static analysis of functional programs

We just showed accuracy of the approximation on a theoretical side. Now we investigate the accuracy on a practical point of view. There is a recent and renewed interest for Data flow analysis of higher-order functional programs [25, 22] that was initiated by [20]. None of those techniques is strategy-aware: on Example 8, they all consider the term $c(a(s(0)), f(n))$ as reachable, though it is not with innermost strategy. Example 8 also shows that this is not the case with innermost completion.

We made an alpha implementation of innermost equational completion. This new version of Timbuk, named TimbukSTRAT, is available at [13] along with several examples. On those examples, innermost equational completion runs within milliseconds. Sets of approximation equations, when needed, are systematically defined using [12]. Roughly, the idea is to define a set $E$ such that the set of equivalence classes of $T(\Sigma)$ w.r.t. $E$ is finite. Now, we show that accuracy of innermost equational completion can benefit to static analysis of functional programs. As soon as one of the analysed functions is not terminating (intentionally or because of a bug), not taking the evaluation strategy into account may result into an imprecise analysis. Consider the following OCaml program:

```
let hd= function x::_ -> x;;           let tl= function _::l -> l;;
let rec delete e l=
 if (l=[]) then [] else if (hd l=e) then tl l else (hd l)::(delete e l);;
```

It is faulty: the recursive call should be `(hd l)::(delete e (tl l))`. Because of this error, any call `(delete e l)` will not terminate if `l` is not empty and `hd l` is not `e`. We can encode the above program into a TRS $R$. Furthermore, if we consider only two elements in lists (`a` and `b`), the language $L$ of calls to `(delete a l)`, where `l` is any non empty list of `b`, is regular. Thus, standard completion can compute an automaton over-approximating $R^*(L)$. Besides, the automaton $\mathcal{AIRR}(R)$ recognising normal forms of $R$ can be computed since $R$ is left-linear. Then, by computing the intersection between the two automata, we obtain the automaton recognising an over-approximation of the set of reachable terms in normal form[4]. Assume that we have an abstract OCaml interpreter performing completion and intersection with $\mathcal{AIRR}(R)$:

```
# delete a [b+];;
-:abst list= empty
```

The result `empty` reflects the fact that the `delete` function does not compute any *result*, *i.e.* it is not terminating on all the given input values. Thus the language of results is empty. Now, assume that we consider calls like `hd(delete e l)`. In this case, any analysis technique ignoring the call-by-value evaluation strategy of OCaml will give imprecise results. This is due to the fact that, for any non empty list `l` starting with an element `e'` different from `e`, `(delete e l)` rewrites into `e'::(delete e l)`, and so on. Thus `hd(delete e l)`, can be rewritten into `e'` with an outermost rewrite strategy. Thus, if we use an abstract OCaml interpreter built on the standard completion, we will have the following interaction:

```
# hd (delete a [b+]);;
-:abst list= b
```

The result provided by the abstract interpreter is imprecise. It fails to reveal the bug in the delete function since it totally hides the fact that the delete function does not terminate! Using innermost equational completion and TimbukSTRAT on the same example would permit to have the expected result which is[5]:

```
# hd (delete a [b+]);;
-:abst list= empty
```

We can perform the same kind of analysis for the program `sum` given in the introduction. This program does not terminate with call-by-value (for any input) but it terminates with call-by-name strategy. Again, strategy-unaware methods cannot show this: there are (outermost) reachable terms that are in normal form: the integer results obtained with a call-by-need or lazy evaluation. An abstract OCaml interpreter unaware of strategies would say:

```
# sum s*(0);;
-:abst nat= s*(0)
```

where a more precise and satisfactory answer would be `-:abst nat= empty`. Using TimbukSTRAT, we can get this answer. To over-approximate the set of results of the function `sum` for all natural numbers `i`, we can start innermost equational completion with the initial regular language $\{sum(s^*(0))\}$. Let $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ with $Q_f = \{q_1\}$ and $\Delta = \{0 \rightarrowtail q_0, s(q_0) \rightarrowtail q_0, sum(q_0) \rightarrowtail q_1\}$ be an automaton recognising this language. Timbuk[13] can compute the automaton $\mathcal{AIRR}(R)$. Innermost equational completion with TimbukSTRAT terminates on an automaton (see [15]) where the only product state labelled by $q_1$ is $\langle q_1, p_{\text{red}}\rangle$. This means that terms of the form $sum(s^*(0))$ have no innermost normal form, i.e. the function `sum` is *not terminating* with call-by-value for all input values. On all

[4] Computing $\mathcal{AIRR}(R)$ and the intersection can be done using Timbuk.

[5] Details in [15]; see files `nonTerm1` and `nonTerm1b` in the TimbukSTRAT distribution at [13].

those examples, we used initial automata $\mathcal{A}$ that were not separating equivalences classes of $E$. On those particular examples the precision of innermost completion was already sufficient for our verification purpose. Yet, if accuracy is not sufficient, it is possible to refine $\mathcal{A}$ into an equivalent automaton separating equivalences classes of $E$, see [11]. When necessary, this permits to exploit the full power of the precision Theorem 36 and get an approximation of innermost reachable terms, as precise as possible, w.r.t. $E$.

On the same example, all aforementioned techniques [25, 22, 20], as well as all standard completion techniques [27, 14, 23], give a more coarse approximation and are unable to prove strong non-termination with call-by-value. Indeed, those techniques approximate all reachable terms, independently of the rewriting strategy. Their approximation will, in particular, contain the integer results that are reachable by the call-by-need evaluation strategy.

## 7 Related work

No tree automata completion-like techniques [10, 27, 3, 14, 23] take evaluation strategies into account. They compute over-approximations of *all* reachable terms.

Dealing with reachable terms and strategies was first addressed in [26] in the exact case for innermost and outermost strategies but only for some restricted classes of TRSs, and also in [9]. As far as we know, the technique we propose is the first to over-approximate terms reachable by innermost rewriting for *any* left-linear TRSs. For instance, Example 8 and examples of Section 6 are in the scope of innermost equational completion but are outside of the classes of [26, 9]. For instance, the `sum` example is outside of classes of [26, 9] because a right-hand side of a rule has two nested defined symbols and is not shallow.

Data flow analysis of higher-order functional programs is a long standing and very active research topic [25, 22, 20]. Used techniques ranges from tree grammars to specific formalisms: HORS, PMRS or ILTGs and can deal with higher-order functions. Higher-order functions are not in the scope of the work presented here, though it is possible with tree automata completion in general [16]. None of [25, 22, 20], takes evaluation strategies into account and analysis results are thus coarse when program execution rely on a specific strategy.

## 8 Conclusion

In this paper, we have proposed a sound and precise algorithm over-approximating the set of terms reachable by innermost rewriting. As far as we know this is the first algorithm solving this problem for any left linear TRS and any regular initial set of terms. It is based on tree automata completion and equational abstractions with a set $E$ of approximation equations. The algorithm also minimises the set of added transitions by completing the product automaton (between $\mathcal{A}_{init}$ and $\mathcal{A}_{\mathcal{IRR}}(R)$). We proposed TimbukSTRAT [13], a prototype implementation of this method.

The precision of the approximations have been shown on a theoretical and a practical point of view. On a theoretical point of view, we have shown that the approximation automaton recognises no more terms than those effectively reachable by innermost rewriting modulo the approximation $E$. On the practical side, unlike other techniques used to statically analyse functional programs [25, 22, 20], innermost equational completion can take the call-by-value strategy into account. As a result, for programs whose semantics highly depend on the evaluation strategy, innermost equational completion yields more accurate results. This should open new ways to statically analyse functional programs by taking evaluation strategies into account.

Approximations of sets of ancestors or descendants can also improve existing termination techniques [17, 24]. In the dependency pairs setting, such approximations can remove edges in a dependency graph by showing that there is no rewrite derivation from a pair to another. Besides, it has been shown that dependency pairs can prove innermost termination [19]. In this case, *innermost* equational completion can more strongly prune the dependency graph: it can show that there is no *innermost* derivation from a pair to another. For instance, on the TRS:

$$\begin{array}{l|l|l}
\mathit{choice}(X,Y) \to X & \mathit{choice}(X,Y) \to Y & \mathit{eq}(s(X),s(Y)) \to \mathit{eq}(X,Y) \\
\mathit{eq}(0,0) \to \mathit{tt} & \mathit{eq}(s(X),0) \to \mathit{ff} & \mathit{eq}(0,s(Y)) \to \mathit{ff} \\
g(0,X) \to \mathit{eq}(X,X) & g(s(X),Y) \to g(X,Y) & f(\mathit{ff},X,Y) \to f(g(X,\mathit{choice}(X,Y)),X,Y)
\end{array}$$

We can prove that any term of the form $f(g(t_1,\mathit{choice}(t_2,t_3)),t_4,t_5)$ cannot be rewritten (innermost) to a term of the form $f(\mathit{ff},t_6,t_7)$ (for all terms $t_i \in \mathcal{T}(\Sigma)$, $i = 1\ldots 7$). This proves that, in the dependency graph, there is no cycle on this pair. This makes the termination proof of this TRS simpler than what AProVE [18] does: it needs more complex techniques, including proofs by induction. Simplification of termination proofs using innermost equational completion should be investigated more deeply.

For further work, we want to improve and expand our implementation of innermost equational completion in order to design a strategy-aware and higher-order-able static analyser for a reasonable subset of a real functional programming language with call-by-value like OCaml, F#, Scala, Lisp or Scheme. On examples taken from [25], we already showed in [16] that completion can handle some higher-order functions. We also want to study if the innermost completion covers the TRS classes preserving regularity of [26, 9], like standard completion does for many decidable classes [8].

Another objective is to extend this completion technique to other strategies. It should be easy to extend those results to the case of *leftmost* or *rightmost* innermost strategy. This should be a simple refinement of the second phase of completion of innermost critical pairs, when supplementary transitions are added. To encode leftmost (resp. rightmost) innermost, for each transition $f(q_1,\ldots,q_{i-1},\langle q,p_{red}\rangle,q_{i+1},\ldots,q_n) \rightarrowtail \langle q'',p''\rangle$, we should add a new transition $f(q_1,\ldots,q_{i-1},\langle q,p_{r\sigma}\rangle,q_{i+1},\ldots,q_n) \rightarrowtail \langle q'',p'''\rangle$, only if all states $q_1,\ldots,q_{i-1}$ (resp. $q_{i+1},\ldots,q_n$) have a $p$ component that is not $p_{red}$. Another strategy of interest for completion is the outermost strategy. This would improve the precision of static analysis of functional programming language using call-by-need evaluation strategy, like Haskell. Extension of this work to the outermost case is not straightforward but it may use similar principles, such as running completion on a pair automaton rather than on single automaton. States in tree automata are closely related to positions in terms. To deal with the innermost strategy, in states $\langle q,p\rangle$, the $p$ component tells us if terms $s$ (or subterms of $s$) recognised by the state $\langle q,p\rangle$ are reducible or not. This is handy for innermost completion because we can decide if a tuple $(\ell \to r,\sigma,\langle q',p'\rangle)$ is an *innermost* critical pair we checking if the $p$ components of the states recognising strict subterms of $\ell\sigma$ are different from $p_{red}$. For the outermost case, this is exactly the opposite: a tuple $(\ell \to r,\sigma,\langle q',p'\rangle)$ is an *outermost* critical pair only if all the *contexts* $C[\,]$ such that $C[\ell\sigma]$ is recognised, are irreducible contexts. If it is possible to encode in the $p'$ component (using an automaton or something else) whether all contexts embedding $\langle q',p'\rangle$ are irreducible or not, we should be able to define outermost critical pairs and, thus, outermost completion in a similar manner.

**Acknowledgements.** The authors thank René Thiemann for providing the example of innermost terminating TRS for AProVE, Thomas Jensen, Luke Ong, Jonathan Kochems, Robin Neatherway and the anonymous referees for their comments.

## References

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

2 Y. Boichut, J. Chabin, and P. Réty. Over-approximating descendants by synchronized tree languages. In *RTA'13*, volume 21 of *LIPIcs*, pages 128–142. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

3 Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Handling non left-linear rules when completing tree automata. *IJFCS*, 20(5), 2009.

4 C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore: a collapsible approach to higher-order verification. In *ICFP'13*. ACM, 2013.

5 G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL'14*. ACM, 2014.

6 H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://tata.gforge.inria.fr`, 2008.

7 H. Comon and Jean-Luc Rémy. How to characterize the language of ground normal forms. Technical Report 676, INRIA-Lorraine, 1987.

8 G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383, 2004.

9 A. Gascon, G. Godoy, and F. Jacquemard. Closure of Tree Automata Languages under Innermost Rewriting. In *WRS'08*, volume 237 of *ENTCS*, pages 23–38. Elsevier, 2008.

10 T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *RTA'98*, volume 1379 of *LNCS*, pages 151–165. Springer, 1998.

11 T. Genet. A note on the Precision of the Tree Automata Completion. Technical report, INRIA, 2014. `https://hal.inria.fr/hal-01091393`.

12 T. Genet. Towards Static Analysis of Functional Programs using Tree Automata Completion. In *WRLA'14*, volume 8663 of *LNCS*. Springer, 2014.

13 T. Genet, Y. Boichut, B. Boyer, V. Murat, and Y. Salmon. Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1. `http://www.irisa.fr/celtique/genet/timbuk/`.

14 T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45:574–597, 2010.

15 T. Genet and Y. Salmon. Reachability Analysis of Innermost Rewriting. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00848260/PDF/main.pdf`.

16 T. Genet and Y. Salmon. Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf`.

17 A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. In *RTA'05*, volume 3467 of *LNCS*, pages 353–367. Springer, 2005.

18 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with aprove. In *IJCAR'14*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.

19 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasonning*, 37(3):155–203, 2006.

20 N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1-3):120–136, 2007.

21 N. Kobayashi. Model Checking Higher-Order Programs. *Journal of the ACM*, 60.3(20), 2013.

22 J. Kochems and L. Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *RTA'11*, volume 10 of *LIPIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

23 A. Lisitsa. Finite Models vs Tree Automata in Safety Verification. In *RTA'12*, volume 15 of *LIPIcs*, pages 225–239, 2012.

24 A. Middeldorp. Approximations for strategies and termination. *ENTCS*, 70(6):1–20, 2002.

25 L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*. ACM, 2011.

26 P. Réty and J. Vuotto. Regular Sets of Descendants by some Rewrite Strategies. In *RTA'02*, volume 2378 of *LNCS*. Springer, 2002.

27 T. Takai, Y. Kaji, and H. Seki. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In *RTA'11*, volume 1833 of *LNCS*. Springer, 2000.

28 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

29 N. Vazou, P. Rondon, and R. Jhala. Abstract Refinement Types. In *ESOP'13*, volume 7792 of *LNCS*. Springer, 2013.