Extending Timbuk to Verify Functional Programs

Thomas Genet, Tristan Gillard, Timothée Haudebourg, Sébastien Lê Cong

Extending Timbuk to Verify Functional Programs

Thomas Genet, Tristan Gillard, Timothée Haudebourg, and Sébastien Lê Cong

Univ Rennes/Inria/CNRS/IRISA, Campus Beaulieu, 35042 Rennes Cedex, France

Abstract. Timbuk implements the Tree Automata Completion algorithm whose purpose is to over-approximate sets of terms reachable by a term rewriting system. Completion is parameterized by a set of equations defining which terms are equated in the approximation. In this paper we present two extensions of Timbuk which permit us to automatically verify safety properties on functional programs. The first extension is a language, based on regular tree expressions, which eases the specification of the property to prove on the program. The second extension automatically generates a set of equations adapted to the property to prove on the program.

1 Motivations

Term Rewriting Systems (TRS for short) are a well known model of functional programs. This model has been used for different kind of analysis ranging from model-checking [4], to static analysis [16] and from termination analysis [13] to complexity analysis [1]. In this paper we focus on static analysis of safety properties on functional programs. Let us illustrate this on a simple example. Assume that we want to analyze the following `delete` OCaml function:

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

In Timbuk [9], this program will be translated in the following TRS, where `ite` encodes the if-then-else construction and `eq` encodes a simple equality on two arbitrary constant symbols `A` and `B`. The **Ops** section defines the symbols with their arity, the **Const** section defines the *constructor* symbols (symbols that are not associated with a function), the **Vars** section defines variables and the **TRS** section associates the name of the TRS with its rules. In the following, we denote by $\mathcal{F}$ the set of symbols defined in the **Ops** section and $\mathcal{T}(\mathcal{F})$ the set of ground terms built on $\mathcal{F}$. We denote by $\mathcal{C}$ the set of constructor symbols defined by **Const**, and $\mathcal{T}(\mathcal{C})$ the set of ground terms defined on $\mathcal{C}$.

```
Ops delete:2 cons:2 nil:0 A:0 B:0 ite:3 true:0 false:0 eq:2
Const A B nil cons true false
Vars X Y Z
TRS R
  delete(X,nil)->nil
```

```
delete(X,cons(Y,Z))->ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
ite(true,X,Y)->X
ite(false,X,Y)->Y
eq(A,A)->true   eq(A,B)->false   eq(B,A)->false   eq(B,B)->true
```

Let us denote by L the set of all possible lists of `A`'s and `B`'s. On this program, we are interested in proving that for all `l` $\in L$, `delete(A,l)` can only result into a list where `A` does not occur. This is equivalent to proving that for all `l`$\in L$, `delete(A,l)` never rewrites to a list containing an `A`. This can be done using reachability analysis on rewriting with the above TRS $\mathcal{R}$. We denote by I the set of all *initial terms*, i.e., $I = \{\texttt{delete}(\texttt{A},\texttt{l}) \mid \texttt{l} \in L\}$ and let Bad be the set of lists containing at least one `A`. We denote by $\mathcal{R}^*(I)$ the set of terms reachable by rewriting terms of I with $\mathcal{R}$, i.e., $\mathcal{R}^*(I) = \{t \mid s \in I \text{ and } s \to_{\mathcal{R}}{}^* t\}$, where $\to_{\mathcal{R}}{}^*$ is the reflexive and transitive closure of $\to_{\mathcal{R}}$. If $\mathcal{R}^*(I) \cap Bad = \emptyset$ then there is no way to rewrite a term of the form `delete(A,l)` with `l`$\in L$ into a list containing an `A` and the property is also true on the functional program. Note that the property proved on the TRS is stronger than the property proved on the functional program. In particular, it is independent of the evaluation strategy: it can be call-by-value as well as call-by-name. Thus, the property is true for OCaml as well as for Haskell programs.[1] This paper presents two extensions of Timbuk making the above analysis possible and automatic.

- The first extension are *simplified regular tree expressions* which let the user easily and intuitively define the set of initial terms I.
- The second extension automatically generates abstraction equations, using algorithms described in [10] and [8]. This makes it possible to automatically build a regular over-approximation App of $\mathcal{R}^*(I)$ such that $App \cap Bad = \emptyset$, if it exists.

In Section 2, we define simplified regular expressions. In Section 3, we explain why abstraction equations are necessary and we show how to generate them in Section 4. In Section 5, we show how to interact with Timbuk in order to carry out a complete analysis, as the one shown above. Finally, in Section 6, we conclude and give further research directions.

2 Simplified regular tree expressions

We defined the TRS but we still need to define the set of initial terms I in Timbuk. Until now, it could only be defined using a tree automaton [5]. Defining I with this formalism is possible but it is error-prone and lacks readability. As in the case of word languages, there exists an alternative representation for regular tree languages: *regular tree expressions* [5]. However, unlike classical regular expressions for words, regular tree expressions are difficult to read and to write. For

[1] When the analysis depends on the evaluation strategy, completion can be extended to take it into account [12].

instance, the regular tree expression defining terms of the form $f(g^n(a), h^m(b))$ with $n, m \in \mathbb{N}$ is $f(g(\Box_1)^{*,\Box_1}.\Box_1 a, h(\Box_2)^{*,\Box_2}.\Box_2 b)$, where $\Box_1$ and $\Box_2$ are new constants. In this expression, the sub-expression $g(\Box_1)^{*,\Box_1}.\Box_1 a$ represents the language $g^n(a)$. The effect of $^{*,\Box_1}$ is to iteratively replace $\Box_1$ by $g(\Box_1)$, and the effect of $.\Box_1 a$ is to replace $\Box_1$ by a. Regular tree expressions are expressive enough to define any regular tree language. To be complete w.r.t. regular tree languages, this formalism needs named placeholders (like $\Box_1$ and $\Box_2$ above) because the effect of the iteration symbol $*$ depends on the position where it occurs. However, named placeholders make regular tree expressions difficult to read and to write, even if they define simple languages. For instance, the set $I = \{\texttt{delete(A,l)} \mid \texttt{l} \in L\}$ defined above can be written $delete(A, cons((A|B), \Box_1)^{*,\Box_1}.\Box_1 nil)$ where $\Box_1$ is a new constant.

In this paper, we propose a new formalism for defining regular tree languages: *simplified regular tree expression* (SRegexp for short). Those expressions are not complete w.r.t. regular languages but are easier to read and to write. For instance, the set I is defined by the SRegexp `delete(A,[cons((A|B),*|nil)])`. Those regular expressions are defined using only 3 operators: '`|`' to build the union of two languages, '`*`' to iterate a pattern and the optional brackets '`[ ... ]`' to define the scope of the embedded `*`. The SRegexp `cons((A|B),*|nil)` repeats the pattern `cons(A, _)` or `cons(B,_)` as long as possible and terminates by `nil`. Thus, it defines the language {`nil, cons(A,nil), cons(B,nil), cons(A,cons(A,nil)), cons(A,cons(B,nil)),`...}. The brackets define the scope of the pattern to repeat with `*`. In the SRegexp `delete(A,[cons((A|B),*|nil)])`, the iteration applies on `cons(A, _)` or `cons(B,_)` but not on `delete(A,_)`. Thus, this expression represents the language {`delete(A, nil), delete(A,cons(A,nil)), delete(A,cons(B,nil)), ...`}.[2]

We implemented SRegexp in Timbuk together with a translation to standard regular tree expressions. We also implemented the translation from regular tree expressions to tree automata defined in [18]. Thus, from a SRegexp I, Timbuk can automatically generate a tree automaton $\mathcal{A}$ whose recognized language $\mathcal{L}(\mathcal{A})$ is equal to I. We also implemented the converse operations: tree automata to regular expression using the algorithm [15] and regular tree expressions to SRegexp. Note that, since SRegexp are incomplete w.r.t. regular tree languages, conversion from regular tree expression to SRegexp may fail. Thus, the over-approximation of reachable terms computed by Timbuk is presented as a SRegexp if it is possible, or as a tree automaton otherwise.

3 The need for abstraction equations

Starting from $\mathcal{R}$ and $I = \mathcal{L}(\mathcal{A})$, computing $\mathcal{R}^*(I)$ is not possible in general [14]. Nevertheless, if $\mathcal{R}$ is a left-linear TRS then $\mathcal{R}^*(I)$ can be over-approximated with tree automata completion [6]. From $\mathcal{A}$ and $\mathcal{R}$, completion builds a tree automaton $\mathcal{A}^*$ such that $\mathcal{L}(\mathcal{A}^*) \supseteq \mathcal{R}^*(I)$. If Bad is regular, to prove $\mathcal{R}^*(I) \cap Bad = \emptyset$,

[2] See the page `http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/simplifiedRegexp.html` for more examples.

it is enough to check that $\mathcal{L}(\mathcal{A}^*) \cap Bad = \emptyset$, which can be done efficiently [5]. For this technique to succeed, the precision of the approximation $\mathcal{A}^*$ is crucial. For instance, $\mathcal{L}(\mathcal{A}^*) = \mathcal{T}(\mathcal{F})$ is a valid regular over-approximation but it cannot be used to prove any safety property since it also contains Bad. In Timbuk, approximations are defined using sets of abstraction equations, following [20] and [11].

Example 1. Let L be the set of terms defined with the symbol s of arity 1 and the constant symbol 0. Let X be a variable. The effect of the equation $s(s(X)) = s(X)$ is to merge in the same equivalence class terms $s(s(0))$ and $s(0)$, $s(s(s(0)))$ and $s(s(0))$, etc. Thus, with this single equation, $L/_{=_E}$ consists of only two equivalence classes: a class containing only 0 and the class containing all the other natural numbers $\{s(0), s(s(0)), \ldots\}$. An equation $s(X) = X$ would define a single equivalence class containing all natural numbers. It would thus define a rougher abstraction. An equation $s(s(X)) = X$ defines two equivalence classes: the class of even numbers $\{0, s(s(0)), \ldots\}$ and the class of odd numbers $\{s(0), s(s(s(0))), \ldots\}$.

For completion to terminate, the set $\mathcal{T}(\mathcal{F})/_{=_E}$ (E-equivalence classes of $\mathcal{T}(\mathcal{F})$) has to be finite [8]. When dealing with functional programs, this restriction can be relaxed as follows. Functional programs manipulate sorted terms and the associated TRSs preserve sorts. Provided that equations also preserve sorts, having a finite set $\mathcal{T}(\mathcal{F})^S/_{=_E}$, where $\mathcal{T}(\mathcal{F})^S$ is the set of well-sorted terms, is enough. Besides, since well-sorted terms define a regular language, this information can be provided to Timbuk using tree automata, regular expressions or SRegexp.

Example 2. Let us consider the set L of well-sorted lists of A and B. The set L is the regular language associated with the SRegexp $cons((A|B), *|nil)$. Let X, Y, Z be variables. The set $E = \{cons(X, cons(Y, Z)) = cons(Y, Z)\}$ defines a set of E-equivalence classes $L/_{=_E}$ with three classes: one class only contains nil, one class contains all lists ending with an A and the last class contains all lists ending with a B.

Going back to the `delete` example that we want to analyze, with set $E = \{cons(X, cons(Y, Z)) = cons(Y, Z)\}$, $L/_{=_E}$ is finite but $\mathcal{T}(\mathcal{F})^S/_{=_E}$ may not be. For instance, terms $delete(A, nil)$, $delete(A, delete(A, nil))$, etc. are all in separate equivalence classes. Again, we can take advantage from the fact that `delete` is a functional program and relax the termination condition of completion by focusing it on the data manipulated by the program. Instead of asking for finiteness of $\mathcal{T}(\mathcal{F})^S/_{=_E}$, we only require finiteness of $\mathcal{T}(\mathcal{C})^S/_{=_E}$, where $\mathcal{T}(\mathcal{C})^S$ is the set of *well-sorted constructor terms*. Let us note E_c the above set of equations $\{cons(X, cons(Y, Z)) = cons(Y, Z)\}$. As shown in Example 2, E_c defines a finite set of equivalence classes on $\mathcal{T}(\mathcal{C})^S$, i.e., lists of A's and B's.[3] Provided that

[3] In fact, in $\mathcal{T}(\mathcal{C})^S$ there are also terms `true` and `false` but they cannot be embedded in lists. Thus, each of them defines its own equivalence class. In the end, in $\mathcal{T}(\mathcal{C})^S/_{=_{E_c}}$ there are 5 equivalence classes.

`delete` is a terminating and complete functional program, it is possible to extend E_c so that completion terminates. This has been shown for first-order functional programs [7] and for higher-order functional programs [10]. The extension of E_c consists in adding two sets of equations $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$ and $E_r = \{f(X_1, \ldots, X_n) = f(X_1, \ldots, X_n) \mid f \in \mathcal{F}$, arity of f is n, and $X_1, \ldots, X_n$ are variables$\}$. Since $E_{\mathcal{R}}$ and E_r are fixed by the program, the precision of the approximation only depends on the equivalence classes defined by E_c. Thus, to explore approximations, it is enough to explore all possible E_c.

4 Generating abstraction equations E_c

Additionally to the fact that (1) ${\mathcal{T}(\mathcal{C})^S}/_{=E_c}$ has to be finite, the termination theorems of [7,10] imposes additional constraints on E_c. Equations in E_c have to be *contracting*, i.e., they are of the form $u = u|_p$ where (2) $u|_p$ is a strict subterm of u and (3) $u|_p$ has the same sort as u.[4] Conditions (2) and (3) makes it possible to prune the search space of equations in E_c. For instance the following equations do not need to be considered: $cons(X, Y) = Z$ because of condition (2), $cons(X, cons(Y, Z)) = cons(X, Z)$ because of condition (2), $cons(X, Y) = X$ because of condition (3).

Timbuk implements two different algorithms to explore the space of possible E_c. Those algorithms are parameterized by a natural number $k \in \mathbb{N}$ and, for a given k, they generate a set $EC(k)$ of possible E_c. By increasing k, we increase the precision of equations sets E_c in $EC(k)$. The first algorithm is based on covering sets [17] and generates contracting equations with variables [10]. In this algorithm k defines the depth of the covering set used to generate the equations. From a covering set S, we generate all equations sets $E_c = \{u = u|_p \mid u \in S\}$ satisfying conditions (1) to (3).

Example 3. Let X be a variable and $\mathcal{T}(\mathcal{C})^S$ be the set of well-sorted constructor terms defined with symbol s of arity 1 and the constant symbol 0. For $k = 1$, the covering set is $\{s(X), 0\}$ and $EC(1) = \{\{s(X) = X\}\}$. For $k = 2$, the covering set is $\{s(s(X)), s(0), 0\}$ and $EC(2) = \{\{s(s(X)) = X\}, \{s(s(X)) = s(X)\}, \{s(0) = 0\}, \{s(0) = 0, s(s(X)) = X\}, \{s(0) = 0, s(s(X)) = s(X)\}\}$.

The second algorithm generates *ground* contracting equations [8]. In this algorithm k represents the number of equivalence classes expected in ${\mathcal{T}(\mathcal{C})^S}/_{=E_c}$. Since equation sets have to be ground and meet conditions (2) and (3), we can finitely enumerate all the possible equations sets E_c for a given k.

Example 4. Let $\mathcal{T}(\mathcal{C})^S$ be the set of well-sorted constructor terms defined with symbol s of arity 1 and the constant symbol 0. For $k = 1$ the set $EC(1) = \{\{s(0) = 0\}\}$. For $k = 2$, the set $EC(2) = \{\{s(s(0)) = 0\}, \{s(s(0)) = s(0)\}$.

[4] Note that the sort information can be inferred from the tree automaton recognizing well-sorted terms. For instance, the automaton associated to the SRegexp of Example 2 recognizes A and B by into the same state, thus A and B will have the same sort (see automaton `TC` in Section 5)

A systematic way to build ground $EC(k)$, based on tree automata enumeration, is given in [8]. Using the first or second algorithm to generate $EC(k)$, to prove that there exists a tree automaton $\mathcal{A}^*$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ and such that $\mathcal{L}(\mathcal{A}^*) \cap Bad = \emptyset$, we run the following algorithm:

1. Start with $k = 1$
2. Build $EC(k)$
3. Pick one E_c in $EC(k)$
4. Complete $\mathcal{A}$ into $\mathcal{A}^*$ using $\mathcal{R}$ and $E_c \cup E_{\mathcal{R}} \cup E_r$
5. If $\mathcal{L}(\mathcal{A}^*) \cap Bad = \emptyset$ then verification is successful
   Otherwise, if $EC(k)$ not empty, pick a new E_c in $EC(k)$ and go to 4.
6. When $EC(k)$ is empty, increment k and go to 2.

It has been shown in [8] that the ground enumeration of $EC(k)$ is complete w.r.t. tree automata that are closed by $\mathcal{R}$-rewriting. Thus, if there exists such a $\mathcal{A}^*$, the above iterative algorithm will find it. However, on properties that cannot be shown using a regular approximation, such as [2], this algorithm may diverge.

5 Interacting with Timbuk

Download `http://people.irisa.fr/Thomas.Genet/timbuk/timbuk3.2.tar.gz` and compile and install Timbuk 3.2. The online version of Timbuk does not integrate all the features presented here. In Timbuk's archive, the full specification of the `delete` example can be found in the file `FunExperiments/deleteBasic.txt`.

```
Ops delete:2 cons:2 nil:0 A:0 B:0 ite:3 true:0 false:0 eq:2
Const A B nil cons true false
Vars X Y Z
TRS R
   delete(X,nil)->nil
   delete(X,cons(Y,Z))->ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
   ite(true,X,Y)->X
   ite(false,X,Y)->Y
   eq(A,A)->true   eq(A,B)->false   eq(B,A)->false   eq(B,B)->true
SRegexp A0
   delete(A,[cons((A|B),*|nil)])
Automaton TC
  States qe ql qb
  Final States qe ql qb
  Transitions
     A->qe  B->qe  nil->ql  cons(qe,ql)->ql  true->qb  false->qb
Patterns
   cons(A,_)
```

This file contains the TRS, the SRegexp presented above and a tree automaton named `TC` which defines well-sorted constructor terms as explained in Example 2. This automaton is used to prune equation generation. Note that this automaton could be inferred from the typing information of the functional program. Here,

the automaton `TC` states that lists are built with `cons` and `nil`, that elements of the list are either `A` or `B`, and that `true` and `false` are of the same type but cannot appear in a list. Thus, ill-typed terms of the form $cons(nil, true)$ are not considered for equation generation. Finally, the `Patterns` section defines the set *Bad* of terms that should not be reachable. Currently, the pattern section is limited to terms or patterns (terms with holes '_') and cannot handle SRegexp or automata. In the present example, we only consider a subset of bad terms: terms of the form `cons(A,_)`, i.e., lists starting by `A`. Assuming that your working directory is `FunExperiments`, you can run Timbuk on this example by typing: `timbuk --fung 30 deleteBasic.txt`. Where `--fung` is the option triggering ground equation generation (the second algorithm for generating $EC(k)$) and `30` is a maximal number of completion steps. We get the following output:

```
Generated equations:
--------------------
cons(A,cons(A,nil)) = cons(A,nil)
cons(B,cons(A,nil)) = cons(A,nil)
cons(B,nil) = nil
B = B
nil = nil
delete(X,Y) = delete(X,Y)
A = A
true = true
cons(X,Y) = cons(X,Y)
false = false
ite(X,Y,Z) = ite(X,Y,Z)
eq(X,Y) = eq(X,Y)
eq(A,A) = true
eq(A,B) = false
eq(B,A) = false
eq(B,B) = true
delete(X,nil) = nil
delete(X,cons(Y,Z)) =
  ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
ite(true,X,Y) = X
ite(false,X,Y) = Y

Regular expression:
-------------------
[cons(B, *|nil)]

Proof done!
-----------
Completion time: 0.006595 seconds
```

The three first generated equations belong to E_c, reflexive equations of the form `B = B`, `nil = nil`, ... belong to E_r and the last eight equations belong to $E_{\mathcal{R}}$. The set $\mathcal{T}(\mathcal{C})^S/_{=E_c}$ has two equivalence classes: the class containing *nil* and all lists containing only B's and the class of lists containing at least one A. Thus, the effect of E_c is to forget any B and preserve any A that appears in a list. Using the `--fun` option instead of `--fung` while running Timbuk, triggers the first algorithm for generating $EC(k)$, i.e., E_c with variables. On this example, the generated E_c part has two equations instead of three: `cons(X,cons(A,Y)) = cons(A,Y)` and `cons(B,X) = X`. The effect of this set E_c is the same as the ground E_c above. Indeed, this E_c splits lists into two equivalence classes: the class of lists without A's and the class of lists with at least one A.

Finally, in Timbuk's output, `Proof done!` means that Timbuk manages to build a regular approximation of $\mathcal{R}^*(I)$ that contains no term of the `Patterns` section. Timbuk outputs the resulting simplified regular expression `[cons(B, *|nil)]`. This proves that results are lists without any occurrence of `A`'s. Here, one can read the outputted SRegexp to check that the property is true. How-

ever, this can be difficult when the outputted SRegexp is more complex. Thus, on most examples, we use additional predicates to check properties like it is commonly done with proof assistants. On our previous example, given a predicate `member` (testing membership on lists), we can check that terms of the language `member(A,delete(A,cons((A|B),*|nil)))` never rewrite to `true`. We can also check the dual property expected on `delete`: deleting A's should not delete all B's. We hope to check this property using initial terms `member(B,delete(A,[cons((A|B),*|nil)]))` and a `patterns` section set to `false`. However, the property is not true and, during completion, Timbuk finds a counterexample:

```
Found a counterexample:
-----------------------
Term member(B,delete(A,nil)) rewrites to a forbidden pattern
```

For the property to hold, lists in initial terms should contain at least one B:
`member(B,delete(A,[cons((A|B),*|[cons(B,*|[cons((A|B),*|nil)])])]))`
Using this initial set of terms, Timbuk succeeds to do the proof and produces a slightly different E_c: `cons(A,cons(B,nil)) = cons(B,nil)`, `cons(B,cons(B,nil)) = cons(B,nil)`, `cons(A,nil) = nil`. This time, E_c forgets about A's and preserves B's. More than 20 other examples (with ground/non-ground equations generation) can be found on the Timbuk page `http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/`, including functions on lists, trees, sorting functions, higher-order functions, etc.

6 Conclusion and further Research

We know that completion is terminating on higher-order functional programs thanks to the recent result of [10]. Besides, we also know that ground equation generation of E_c is complete w.r.t. tree automata that are closed by $\mathcal{R}$ [8]. In other words, if there exists a tree automaton $\mathcal{A}^*$, closed by $\mathcal{R}$ and over-approximating the set of reachable terms, then it will eventually be found by generating ground equations. With the first algorithm where equations of E_c may contain variables, we do not have a similar completeness result, yet. However, generating equations with variables remains an interesting option because the set E_c can be smaller. This is the case in the previous example where E_c with variables defines the same set of equivalence classes but with fewer equations.

From a theoretical perspective, Tree Automata Completion can be seen as an alternative to well-established higher-order model-checking techniques like PMRS [21] or HORS [19] to verify higher-order functional programs. Timbuk implements Tree Automata Completion but was missing several features for those theoretical results to be usable in practice. First, stating the property to prove using a tree automaton was error-prone and lacked readability. Using simplified regular expressions significantly improves this step and makes property definition closed to what is generally used in a proof assistant. Second, equations which are necessary to define the approximation, had to be given by the user [7]. Now, Timbuk can automatically generate a set of equations adapted to

a given verification objective. Combining those two extensions makes Timbuk a competitive alternative to higher-order model checking tools like [21] and [19].

In those model-checking tools and in Timbuk, the properties under concern are "regular properties", i.e. properties proven on regular languages. Those regular properties are stronger than what offers tests (they prove a property on an infinite set of values) but weaker than what can be proven using induction in a proof assistant. However, unlike proof assistants, Timbuk does not require to write lemmas or proof scripts to prove a regular property. An interesting research direction is to explore how to lift those regular properties to general properties. In other words, how to build a proof that $\forall$ `x l. not(member(x,delete(x,l)))` from the fact that all terms from `member(A,delete(A,cons((A|B),*|nil)))` rewrite to `false`. We believe that this is possible by taking advantage of parametricity such as in [22]. This is ongoing work.

In this paper, the verification is performed on a TRS representing the functional program. To directly perform the verification on real functional programs rather than on TRSs, we need a transformation. We could reuse the HOCA transformation of [1]. However, it does not take the priorities of the pattern matching rules of the functional program into account when producing the TRS. Furthermore, this translation needs to be certified, i.e., we need a formal proof that the behavior of the outputted TRS $\mathcal{R}$ covers all the possible behaviors of the functional program. With such a proof on $\mathcal{R}$, if Timbuk can prove that no term of `member(A,delete(A,cons((A|B),*|nil)))` can be rewritten to `true` with $\mathcal{R}$, then we have a similar property on the functional program.

The equation generation process does not cover all TRSs but only TRSs encoding terminating, complete, higher-order, functional programs. We currently investigate how to generate equations without the termination and completeness restrictions on the program. Another research direction is to extend this verification principle to more general theorems. For the moment, theorems that can be proved using Timbuk need to have a regular model. For instance, Timbuk is able to prove the theorem `member(A,delete(A,l))` $\not\rightarrow_{\mathcal{R}}{}^{*}$ `true` for all lists `l=cons((A|B),*|nil)` because the language of terms reachable from the initial language `member(A,delete(A,cons((A|B),*|nil)))` is, itself, regular. Assume that we have a predicate `eq` encoding equality on lists. To prove a theorem of the form `eq(delete(A,l),l)` $\not\rightarrow_{\mathcal{R}}{}^{*}$ `false` for all list `l=cons(B,*|nil)`, the language of reachable terms is no longer regular. However, recent advances in completion-based techniques for non-regular languages [3] should make such verification goals reachable.

Acknowledgements

Many thanks to the anonymous referees for their valuable comments.

References

1. M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *ICFP'15*, pages 152–164. ACM, 2015.

2. Y. Boichut and P.-C. Héam. A theoretical limit for safety verification techniques with regular fix-point computations. *IPL*, 108(1):1–2, 2008.
3. Yohan Boichut, Jacques Chabin, and Pierre Réty. Towards more precise rewriting approximations. In *LATA'15*, volume 8977 of *LNCS*, pages 652–663. Springer, 2015.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
5. 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.
6. 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.
7. T. Genet. Termination Criteria for Tree Automata Completion. *Journal of Logical and Algebraic Methods in Programming*, 85, Issue 1, Part 1:3–33, 2016.
8. T. Genet. Automata Completion and Regularity Preservation. Technical report, INRIA, 2017. `https://hal.inria.fr/hal-01501744`.
9. T. Genet, Y. Boichut, B. Boyer, T. Gillard, T. Haudebourg, and S. Lê Cong. Timbuk 3.2 – a Tree Automata Library. IRISA / Université de Rennes 1, 2017. `http://people.irisa.fr/Thomas.Genet/timbuk/`.
10. T. Genet, T. Haudebourg, and T. Jensen. Verifying higher-order functions with tree automata. In *FoSSaCS'18*, LNCS. Springer, 2018. To be published.
11. T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45:574–597, 2010.
12. T. Genet and Y. Salmon. Reachability Analysis of Innermost Rewriting. In *RTA'15*, volume 36 of *LIPIcs*, Warshaw, 2015. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
13. J. Giesl. Termination Analysis for Functional Programs using Term Orderings. In *SAS'95*, volume 983 of *LNCS*, pages 154–171. Springer, 1995.
14. R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–175, 1995.
15. Y. Guellouma, L. Mignot, H. Cherroun, and D. Ziadi. Construction of rational expression from tree automata using a generalization of Arden's lemma. *CoRR*, abs/1501.07686, 2015.
16. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1-3):120–136, 2007.
17. E. Kounalis. Testing for the Ground (Co-)Reducibility Property in Term-Rewriting Systems. *TCS*, 106(1):87–117, 1992.
18. D. Kuske and I. Meinecke. Construction of tree automata from regular expressions. *RAIRO - Theor. Inf. and Applic.*, 45(3):347–370, 2011.
19. Y. Matsumoto, N. Kobayashi, and H. Unno. Automata-Based Abstraction for Automated Verification of Higher-Order Tree-Processing Programs. In *APLAS'15*, volume 9458 of *LNCS*, pages 295–312. Springer, 2015.
20. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *TCS*, 403(2-3):239–264, 2008.
21. L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*. ACM, 2011.
22. Philip Wadler. Theorems for free! In *Proc. of FPCA'89*, pages 347–359. ACM, 1989.