Regular Language Type Inference with Term Rewriting - extended version

Thomas Haudebourg, Thomas Genet, Thomas Jensen

Regular Language Type Inference with Term Rewriting – extended version

TIMOTHÉE HAUDEBOURG, Univ Rennes, Inria, CNRS, IRISA, France
THOMAS GENET, Univ Rennes IRISA, France
THOMAS JENSEN, Inria, France

This paper defines a new type system applied to the fully automatic verification of safety properties of tree-processing higher order functional programs. We use term rewriting systems to model the program and its semantics and tree automata to model algebraic data types. We define the regular abstract interpretation of the input term rewriting system where the abstract domain is a set of regular languages. From the regular abstract interpretation we derive a type system where each type is a regular language. We define an inference procedure for this type system which allows us check the validity of safety properties. The inference mechanism is built on an invariant learning procedure based on the tree automata completion algorithm. This invariant learning procedure is regularly-complete and complete in refutation, meaning that if it is possible to give a regular type to a term then we will eventually find it, and if there is no possible type (regular or not) then we will eventually find a counter-example.

1 INTRODUCTION

The spectrum of type systems for verifying higher-order functional program is wide. At one end we have the fully automatic type systems with complete and efficient type inference algorithms but of limited expressiveness. At the other end we find type systems like F* [Microsoft Research and Inria 2013], Liquid Types [Rondon et al. 2008] or Bounded Refinement Types [Vazou et al. 2015, 2013], and proof assistants such as Coq [Inria 2016] or Isabelle/HOL [Nipkow et al. 2002] with an expressive set of types but which require (expert) user interaction in the form of type annotations or definition of intermediate lemmas.

In this paper, we explore the middle ground between these extremes and propose a verification technique for higher-order functional programs that is both expressive and has fully automatic inference. We focus on verifying properties on programs concerning *algebraic data types*. To do this, we define an expressive type system based on *regular language types* where types represent regular sets of terms. This extended type system is efficient and powerful enough to prove non-trivial properties on higher-order functional programs, yet it has an automatic inference algorithm. We emphasize that we are interested in *fully automated* verification approaches, accessible to any programmer, where properties are proved without annotations or intermediate lemmas. The price to pay for retaining expressiveness is that termination is only guaranteed for regular properties.

The use of regular languages for the verification of functional programs can be traced back to Jones [Jones 1987; Jones and Andersen 2007] who proposed to model higher-order functions as term rewriting systems and use regular grammars to approximate their result, making it a good candidate for *fully automated* reachability analysis. The precision of the approximations in [Jones and Andersen 2007] was fixed but since then more flexible techniques have been developed. In addition, model checking with abstraction refinement has been used by Ong *et al.* [Ong 2006], Kobayashi *et al.* [Kobayashi 2009; Kobayashi et al. 2011b] to analyse higher-order functions automatically. This has been extended with regular language abstractions to handle abstract data types by Kobayashi *et al.* [Matsumoto et al. 2015] and Genet *et al.* [Genet et al. 2018]. Their approach relies on abstractions for computing over-approximations of the set of reachable states, on which

Authors' addresses: Timothée Haudebourg, timothee.haudebourg@irisa.fr, Univ Rennes, Inria, CNRS, IRISA, Rennes, France; Thomas Genet, thomas.genet@irisa.fr, Univ Rennes, IRISA, Rennes, France; Thomas Jensen, thomas.jensen@inria.fr, Inria, Rennes, France.

safety properties can then be verified. The precision of the approximation is automatically adapted to the property to prove.

In the new verification technique presented in this paper, functions are modeled by term rewriting systems (following [Jones and Andersen 2007]) and properties are verified using regular abstractions whose precision is automatically adapted to the property to verify (as suggested by Matsumoto *et al.* [Matsumoto et al. 2015]). A function is associated with its regular abstraction through a *regular language type*. For a given function, a regular language type annotation gives the possible regular abstractions of its result for the regular abstractions of its input. Combining those aspects leads to a fully automatic verification technique for higher-order functional programs that is particularly well suited to deal with algebraic data types. Properties are defined using predicates expressed in the functional programming language itself. The salient feature of this technique is that it does not rely on the programmer to provide any intermediate lemmas (like in proof assistants) or type annotations (like in Liquid types or F*). We successfully implemented and tested our technique on examples taken from the benchmark suite of Timbuk. The benchmarks also include several verification problems (such as proving that merge sort actually returns a sorted list) that we perform completely automatically whereas they would require intermediate lemmas in proof assistants or additional type annotations in Liquid types or F*.

Representation of higher-order programs as TRS. A core feature of our method is to abstract the values manipulated by functions using regular tree languages. This is close to [Genet et al. 2018; Matsumoto et al. 2015]. We will be using the general formalism of term rewriting systems (TRS) to represent programs. The combination of regular languages and TRS is natural: Algebraic data types are readily modeled by regular languages and many functional programming languages such as OCaml, Haskell can be translated into first-order *applicative* term rewriting systems [Kennaway et al. 1996], as illustrated by the following simple example involving the *filter* function.

```
let rec filter p = function
| [] -> []
| x::l -> if p x
then x::(filter p l)
else filter p l
```

(a) OCaml version

$$
\begin{aligned}
filter\ \ p\ \ nil &\to nil \\
filter\ \ p\ \ cons(x, l) &\to if(p\ \ x, \\
&\qquad cons(x, filter\ \ p\ \ l), \\
&\qquad filter\ \ p\ \ l)
\end{aligned}
$$

(b) Term rewriting system

Fig. 1. Translation of a “filter” function written in OCaml into a term rewriting system

Verification as a reachability problem. To illustrate our approach, assume that we want to verify that the *filter* function of Fig. 1 applied with the *even* predicate always return lists of even numbers. In our framework where the execution model is term rewriting, this property becomes:

$$\forall l.\ \mathit{for\_all}\ \mathit{even}\ (\mathit{filter}\ \mathit{even}\ l) \not\to_{\mathcal{R}}^{*} \mathit{false}$$

where $\to_{\mathcal{R}}$ denotes rewriting w.r.t. a TRS $\mathcal{R}$ defining functions *for_all*, *even*, and *filter*. Proving this can be done by computing the set of *reachable terms* (i.e. terms reachable by rewriting) from the initial set of terms

$$I = \{\ \mathit{for\_all}\ \mathit{even}\ (\mathit{filter}\ \mathit{even}\ l) \mid l \in L\}$$

where L is the set of all lists (*i.e.*, the regular language derived from the definition of the OCaml algebraic data type definition `int list`). Once the set of *reachable terms* is computed, we can prove the property by checking that the set of unwanted terms/values, here { *false* }, does not intersect the set of reachable terms.

Reachable terms are uncomputable in general, but can be abstracted. Since the set L of all lists (and thus I) is infinite, the set of *reachable terms* is *in general* uncomputable. In this paper we abstract those *infinite* sets of reachable terms into regular languages, represented by tree automata. The core of our verification technique is *regular abstract interpretation*, i.e., a special case of abstract interpretation where

- the concrete domain is a set of terms,
- the abstract domain $\Sigma^\#$ is a set of abstract symbols that each represent a regular language of concrete terms,
- the abstraction function is defined as a rewriting system that rewrites a concrete term to an abstract term,
- the abstract semantics of the program is defined by a TRS $\mathcal{R}^\#$, extracted from $\mathcal{R}$, that rewrites abstract terms.

The verification then proceeds by computing with $\mathcal{R}^\#$ on abstract terms that represent regular sets. Referring back to the example from above, the previous property can be *abstracted* as follows:

$$(1) \quad \textit{for\_all even } (\textit{filter even } l^\#) \not\to^*_{\mathcal{R}^\#} \textit{false}^\#$$

where $l^\#$ is an abstract value of $\Sigma^\#$ representing all the well-formed natural number lists, and $\textit{false}^\#$ the abstract value representing the regular language $\{\textit{false}\}$. Since $l^\#$ abstracts all lists, note that the $\forall$ quantifier has disappeared from the abstract version of the property. For a given $\Sigma^\#$ and $\mathcal{R}^\#$, this abstract property can be efficiently checked using well-known dedicated algorithms on TRSs and tree automata [Genet et al. 2018]. However, not all abstractions $\Sigma^\#$ and $\mathcal{R}^\#$ will have that property. The difficult part thus becomes to find a correct abstract domain $\Sigma^\#$ and an abstract TRS $\mathcal{R}^\#$ that can prove the abstract property.

Regular language types for modularizing the inference of $\Sigma^\#$ *and* $\mathcal{R}^\#$. In [Genet et al. 2018; Matsumoto et al. 2015], inferring $\Sigma^\#$ and $\mathcal{R}^\#$ is done for the whole TRS $\mathcal{R}$, at once. In this paper, we aim at modularizing the inference, function by function. We thus need a way to associate a function symbol to all its possible regular language abstractions. We do this using a dedicated type system and dedicated type annotations called *regular language types*. In this system, base types are elements of $\Sigma^\#$ and variables and terms can be annotated using elements of $\Sigma^\#$. For instance, $X : l^\#$ denotes a variable whose value belongs to the regular language $l^\#$, i.e., lists of natural numbers. Assume that $\Sigma^\#$ contains the abstract terms $\textit{even\_list}^\#$ (the regular language of lists of *even* natural number) and $\textit{true}^\#$ (the language $\{\textit{true}\}$). Then, to prove property (1) we can *separately* prove that the functions *filter* and *for_all* have the following types:

$$\textit{filter even } (X : l^\#) : \textit{even\_list}^\# \qquad \textit{for\_all even } (Y : \textit{even\_list}^\#) : \textit{true}^\#$$

where the first part means that $\textit{filter even } l^\#$ can only be rewritten by $\mathcal{R}^\#$ into a list of even numbers and the second part that $\textit{for\_all even even\_list}^\#$ can only be rewritten to $\textit{true}^\#$ (and thus not to $\textit{false}^\#$, provided that $\textit{true}^\#$ and $\textit{false}^\#$ are disjoint languages). One difficulty remains: how do we discover that we need the abstract element $\textit{even\_list}^\#$ to perform this proof? We propose to perform inference of $\Sigma^\#$ and $\mathcal{R}^\#$ function by function and in a goal-directed manner. For property (1) to hold we first need to find the type T for $\textit{for\_all even } (Y : T) : \textit{true}^\#$ to hold. Our type inference algorithm will provide the abstract term (and language) $\textit{even\_list}^\#$ as a solution for T. Then, we type check the last part, i.e., $\textit{filter even } (X : l^\#) : \textit{even\_list}^\#$, which concludes the proof.

1.1 Contributions

This paper presents a new verification technique for higher-order functional programs. The verification technique is fully automatic and can prove properties without type annotation of intermediate functions. This technique focuses on a precise analysis of abstract datatypes using regular languages. The state-of-the-art in regular language inference for higher-order functional program verification consists of two works [Genet et al. 2018] and [Matsumoto et al. 2015]. Our contribution is:

- On the conceptual level, our framework integrates the TRS-based analysis of [Genet et al. 2018] with the type inference approach described in [Matsumoto et al. 2015].
- Our technique is complete on regular problem instances, and complete in refutation (even on non-regular verification problems). This is not the case for [Genet et al. 2018]. Our technique is also more efficient in practice than [Genet et al. 2018].
- Our approach is modular, and optimized for non-recursive functions, which is not the case of [Matsumoto et al. 2015].
- We provide an open, public implementation and a public test suite.

1.2 Paper summary

The rest of the paper is structured as follows. Section 2 gives an informal survey of the technique. Section 3 recalls the baseline definitions of terms, rewriting systems and tree automata used throughout this paper. Section 4 introduces our *regular abstract interpretation* framework. Section 5 presents the regular language type system and the type inference algorithm. A first part is dedicated to the straightforward inference of non-recursive functions types, and a later part to our invariant learning procedure for recursive functions. Section 6 exposes the results of our experiments carried on a custom implementation of the technique while Section 7 compares these results with related work. Finally, Section 8 concludes the paper.

2 OVERVIEW OF THE APPROACH

The section provides an informal overview of our verification technique, before presenting the technical details in the following sections. First, we explain *regular abstract interpretation* and how it can be used to verify a property on a program. Then, we show how to abstract an input program and infer an abstract domain able to verify the property. Finally, we present how to modularize the procedure using *regular language types* for scalability.

2.1 Regular abstract interpretation as an abstraction framework

In the introduction, we used the following property as an example:

$$\forall L.\ \mathit{for\_all\ even\ (filter\ even\ l)} \ \not\to^*_{\mathcal{R}} \ \mathit{false}$$

where $\mathcal{R}$ contains the rewriting rules defining the functions *for_all*, *filter* and *even*. Here, for the sake of simplicity, we focus on a smaller TRS $\mathcal{R}$:

$$\mathit{even}\ 0 \to \mathit{true} \qquad \mathit{even}\ s(X) \to \mathit{odd}\ X \qquad \mathit{odd}\ 0 \to \mathit{false} \qquad \mathit{odd}\ s(X) \to \mathit{even}\ X$$

and the following sub-property:

$$\forall n \in \mathbb{N}.\ n\%2 = 0 \Rightarrow \mathit{even}\ n \ \not\to^*_{\mathcal{R}} \ \mathit{false}$$

As showed earlier, directly verifying this property implies computing the set of terms reachable from the initial language $I = \{\ \mathit{even}\ n \mid n \in \mathit{Even}\ \}$ where $\mathit{Even} = \{n \mid n \in \mathbb{N}, n\%2 = 0\ \}$ is the set of even numbers. We write $\mathcal{R}^*(I)$ for the set of reachable terms. Since I is an unbounded set, $\mathcal{R}^*(I)$ is unbounded and thus uncomputable. However, the language *Even* is a regular language, thus we can represent it finitely using a tree automaton. Tree automata will be formally defined Section 3.

Informally, a tree automaton for the *Even* language is a TRS $\Delta^\#$ rewriting even numbers to the *abstract value* $0^\#$:

$$0 \to 0^\# \qquad\qquad s(0^\#) \to 1^\# \qquad\qquad s(1^\#) \to 0^\#$$

This TRS defines an *abstraction* for every even number. In fact, $1^\#$ also gives an abstraction for odd numbers and, thus, the TRS defines an abstraction for *all* natural number. From an abstract interpretation point of view, this is the role of the abstraction function α: here we have $\alpha(n) = 0^\# \iff n \to^*_{\Delta^\#} 0^\#$ and a similar property for $1^\#$. Thus, $\Delta^\#$ defines how concrete values (terms) are rewritten into their abstraction in the abstract domain $\Sigma^\# = \{0^\#, 1^\#\}$. This shows how TRSs and regular languages are combined to define a *regular abstract interpretation.*

In this framework, our goal is to build, or infer, $\Delta^\#$ along with an abstraction of $\mathcal{R}$ able to verify the desired property. This abstraction of $\mathcal{R}$, named $\mathcal{R}^\#$, is a new TRS defined over the abstract domain such that for all terms t and u:

$$t \to^*_{\mathcal{R}} u \iff \alpha(t) \to^*_{\Delta^\# \cup \mathcal{R}^\#} \alpha(u).$$

Here, $\Delta^\#$ gives the abstraction of the values and $\mathcal{R}^\#$ gives the abstraction of the functions. In our example, a good candidate for $\mathcal{R}^\#$ would include the following two rules

$$\mathit{even}\ 0^\# \to \mathit{true}^\# \qquad\qquad \mathit{even}\ 1^\# \to \mathit{false}^\#$$

where $\Sigma^\#$ has been extended with the two abstract values $\mathit{true}^\#$ and $\mathit{false}^\#$ such that $\mathit{true} \to_{\Delta^\#} \mathit{true}^\#$ and $\mathit{false} \to_{\Delta^\#} \mathit{false}^\#$. In this case, our property $\mathit{even}\ n \not\to^*_{\mathcal{R}} \mathit{false}$ for all even numbers n is verified by the fact that $\mathit{even}\ 0^\# \not\to^*_{\Delta^\# \cup \mathcal{R}^\#} \mathit{false}^\#$.

2.2 Abstractions inference procedure

As stated above, our goal is to infer $\Delta^\#$ and $\mathcal{R}^\#$ for a given program $\mathcal{R}$ to verify a given property. This can be challenging, especially when the input TRS $\mathcal{R}$ contains mutually recursive rules. In this case, we need to infer abstract values representing the regular language invariants of the program. In Section 5.4 we describe a procedure for learning such invariants.

The core of the invariant inference procedure is a counter-example based abstraction refinement procedure, based on the Tree Automaton Completion Algorithm [Genet 2016] and SMT constraints solving. For a given symbol f and target partition P (a set of disjoint languages), our procedure is able to give, for all $L \in P$, all the input regular languages $L_1, \dots L_n$ such that , for all $t_i \in L_i$, $f\ t_1\ \dots\ t_n$, rewrites into a term belonging to L. In our previous example, by giving $\mathcal{R}$, *even* and the target partition $\{\{true\}, \{false\}\}$ to our procedure, it will automatically infer that the interesting input languages for *even* are *Even* and *Odd* (where $Odd = \mathbb{N} \backslash Even$), that $\forall n \in Even.\ even\ n \to^* true$, and that $\forall n \in Odd.\ even\ n \to^* false$. In practice, the target partition is given in terms of *abstract values* in a input abstract domain. In our example, the input partition would be $\{\mathit{true}^\#, \mathit{false}^\#\}$. Similarly, the output of the invariant inference procedure is a refinement of the abstract domain (rules to add to $\Delta^\#$) and an abstract TRS $\mathcal{R}^\#$. In our example, the output of the procedure would be:

$$\begin{array}{cccc}
0 \to 0^\# & s(0^\#) \to 1^\# & s(1^\#) \to 0^\# & \text{to add to } \Delta^\# \\
 & \mathit{even}\ 0^\# \to \mathit{true}^\# & \mathit{even}\ 1^\# \to \mathit{false}^\# & \text{rules of } \mathcal{R}^\# \\
 & \mathit{odd}\ 1^\# \to \mathit{true}^\# & \mathit{odd}\ 0^\# \to \mathit{false}^\# &
\end{array}$$

Note that it also gives an information about the *odd* function, since the definition of *even* is mutually recursive with *odd*. In Section 5.4 we show that this procedure is *regularly complete* and *complete in refutation.* This means that, if there exists a regular abstraction ($\Delta^\#$ and $\mathcal{R}^\#$) that satisfies the input partition, then the procedure will find it. Besides, if there exists an input term rewriting to two abstract values located in two different parts of the input partition P, the procedure terminates, and provides such a counter-example term.

2.3 Adding modularity using a type system of regular languages

The invariant learning procedure can theoretically analyze an entire TRSs at once, directly proving or disproving the input property. However, since both the Tree Automata Completion Algorithm and SMT constraint solving can be expensive on large instances, it is preferable to split the verification problem into smaller instances to increase scalability.

One can observe that the information given by the invariant learning procedure in $\mathcal{R}^\#$ can be seen as a typing environment for the symbols *even* and *odd*. This environment says that for a parameter of type $0^\#$ the *even* function gives a result of type $\mathit{true}^\#$. [1] From this point of view, our types represent regular languages, and a term has a given type if it is rewritten by $\Delta^\#$ and $\mathcal{R}^\#$ to the corresponding language. Thus, the abstractions $\Delta^\#$ and $\mathcal{R}^\#$ are used to type terms. For instance, using our above example, we have $s(s(0)) : 0^\#$ and $\mathit{even}\ s(s(0))) : \mathit{true}^\#$.

In this paper, we pursue this typing perspective by defining a formal type system of regular language types, expressing our verification procedure as a type inference procedure. We define a typing judgment $\Delta^\#, \mathcal{R}^\# \vdash t : \tau$ having the following property:

$$t \to^*_{\Delta^\# \cup \mathcal{R}^\#} \tau \quad \Longleftrightarrow \quad \Delta^\#, \mathcal{R}^\# \vdash t : \tau$$

This allows us to replace every inductive reasoning on abstract rewriting paths (on the left) by an inductive reasoning on the structure of terms (on the right) following the typing rules. By doing so, we can clearly separate the analysis of the different functions of the program.

For instance, consider our previous property:

$$\forall L.\ \mathit{for\_all}\ \mathit{even}\ (\mathit{filter}\ \mathit{even}\ l) \not\to_{\mathcal{R}} \mathit{false}$$

This is now equivalent to inferring all the types of the partially typed pattern

$$\mathit{for\_all}\ \mathit{even}\ (\mathit{filter}\ \mathit{even}\ (L\ :\ \mathit{list})) : \mathit{true}^\#$$

where $\mathit{true}^\#$ serves as a type. The type inference procedure defined in this paper works backward from the target type $\mathit{true}^\#$ and proceeds inductively on the structure of the given pattern, analyzing every fixpoint separately. First, we analyse the *for_all* function to find all the possible input types (τ_1, τ_2) such that $\mathit{for\_all}\ (P : \tau_1)\ (L : \tau_2)\ : \mathit{true}^\#$. We then inductively propagate the result into our initial pattern, analyzing *even* with the target type τ_1 and filter with the target type τ_2, etc. In addition to facilitating the modularization, we believe that using types makes this verification technique easier to use for users familiar with typing system. This also opens up the possibility of letting users add *optional* regular language type annotations to the program both to refine the specification of functions and to speed up the verification procedure.

3 PRELIMINARIES

Our work is based on Term Rewriting Systems (TRS) and Tree Automata as they provide a natural representation of functional programs and algebraic data types, respectively. In particular, TRSs offer a straightforward way to handle pattern-matching, they can be transformed to cover both higher-order functions (see Section 3.2) and beta-reduction of anonymous functions. Note that, in this paper, we do not detail the later transformation called lambda-lifting but it is commonly used in program verification [Johnsson 1985]. This section provides the basic definitions of TRS and

[1] The typing information inferred can be viewed as a kind of intersection types [Rocca 1988]. For instance, the type for *even* could be summarized as $\mathit{even}\ :\ 0^\# \to \mathit{true}^\# \wedge 1^\# \to \mathit{false}^\#$. We shall not pursue the relationship to intersection types further, and will keep using abstract TRSs to represent typing environments.

automata [Baader and Nipkow 1998; Comon et al. 2007], along with a description of the sub-family of TRS we consider in this paper, and how these TRS can be used to model higher-order functional programs.

3.1 Terms, Rewriting Systems and Tree Automata

Alphabets, Patterns and Terms. A ranked alphabet Σ is a finite set of symbols provided with an arity function $ar : \Sigma \to \mathbb{N}$. Symbols are used to represent constructors such as *nil* or *cons*, or functions such as *filter*, etc. For simplicity, we write $f \in \Sigma^n$ when $f \in \Sigma$ and $ar(f) = n$. For a given ranked alphabet Σ and finite set of variables $\mathcal{X}$, the set of *terms*, $\mathcal{T}(\Sigma, \mathcal{X})$, is defined as the smallest set such that:

$$\begin{aligned} x \in \mathcal{T}(\Sigma, \mathcal{X}) &\Leftarrow x \in \mathcal{X} \\ f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X}) &\Leftarrow f \in \Sigma^n \text{ and } t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X}) \end{aligned}$$

A term t is *closed* if it contains no variables. The set of closed terms is written $\mathcal{T}(\Sigma)$. A term is *linear* if the multiplicity of each contained variable is at most 1. A *position* in a term t is a word over $\mathbb{N}$ referencing a *subterm* of t. $Pos(t)$ is the set of positions in t, one for each of its subterm.

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

where λ is the empty word and "." in $i.p$ is the *concatenation* operator. For $p \in Pos(t)$, we write $t|_p$ for the subterm of t at position p, and $t[s]_p$ for the term t where the subterm at position p has been replaced by s. A *substitution* σ is an application of $\mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$, mapping variables to terms. We tacitly extend it to the endomorphism $\sigma : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{T}(\Sigma, \mathcal{X})$ where $t\sigma$ is the result of the application of the substitution σ to the term t. We write $Dom(\sigma) = \mathcal{X}$ the domain of the substitution. In this paper, we refer to terms containing variables as "patterns", and closed terms as simply "terms". A *language* is a set of (closed) terms. If $\mathcal{L}$ is a language, we write $\overline{\mathcal{L}} = \mathcal{T}(\Sigma) \setminus \mathcal{L}$ for the *complement* of $\mathcal{L}$.

Term Rewriting Systems. We use term rewriting systems (TRS) to model functional programs and their semantics. A TRS is a pair $\langle \mathcal{T}(\Sigma, \mathcal{X}), \mathcal{R} \rangle$, where Σ is an alphabet and $\mathcal{R}$ a set of rewriting rules of the form $l \to r$, where $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$. A rewriting rule $l \to r$ is said to be left-linear if the pattern l is linear. From a functional program point of view, each rule of a TRS can be seen as defining one step of computation. We write $\mathcal{R}$ the rewriting system $\langle \mathcal{T}(\Sigma, \mathcal{X}), \mathcal{R} \rangle$ when there is no ambiguity on Σ and $\mathcal{X}$. A rewriting system $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ where for all $s, t \in \mathcal{T}(\Sigma, \mathcal{X})$, $s \to_{\mathcal{R}} t$ if there exists a rule $l \to r \in \mathcal{R}$, a position $p \in Pos(s)$ and a substitution σ such that $l\sigma = s|_p$ and $t = s[r\sigma]_p$. The reflexive-transitive closure of $\to_{\mathcal{R}}$ is written $\to^*_{\mathcal{R}}$. A closed term t is in *normal form* if no rule applies on it. We name $GNF(\mathcal{R})$ the set of ground normal forms of $\mathcal{R}$. We write $\overrightarrow{t}^!$ for the term $u \in GNF(\mathcal{R})$ such that $t \to^*_{\mathcal{R}} u$, if it is unique. We write $\mathcal{R}^*(\mathcal{L}) = \{t \mid s \in \mathcal{L}, s \to^* t\}$ for the set of *reachable terms* from $\mathcal{L}$ using $\mathcal{R}$.

Tree Automata. We use tree automata to represent regular sets of terms, or *regular languages.* A tree automaton $\mathcal{A}$ is a quadruple $\langle \Sigma, Q, Q_f, \Delta \rangle$ where Σ is a ranked alphabet, Q a finite set of states, Q_f the set of *final states*, and Δ a rewriting system over $\mathcal{T}(\Sigma, Q)$. Rules in Δ, called *transitions*, are of the form $l \to q$ where $q \in Q$ and $l \in \mathcal{T}(\Sigma, Q)$, where l is called a "configuration". An *ϵ-transition* is a transition $q' \to q$ where $q' \in Q$. A term t is *recognized* by a state $q \in Q$ if $t \to^*_{\Delta} q$, which we also write $t \to^*_{\mathcal{A}} q$. We write $\mathcal{L}(\mathcal{A}, q)$ for the language of all terms recognized by q. A term t is recognized by the tree automaton $\mathcal{A}$ if there exists $q \in Q_f$ s.t. $t \in \mathcal{L}(\mathcal{A}, q)$. We name $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$. A tree automaton $\mathcal{A}$ is *ϵ-free* if it contains no ϵ-transitions. It is *deterministic* if for all terms t there is at most one state q such that $t \rightarrow_{\Delta}^{*} q$. It is *reduced* if for all q there is at least one term t such that $t \rightarrow_{\Delta}^{*} q$. It is *normalized* if every configuration of the automaton is of the form $f(q_1, \ldots, q_n)$ or q' where $q_1 \ldots q_n, q' \in Q$. It is *complete* if for every term of $\mathcal{T}(\Sigma)$, there exists a state in $\mathcal{A}$ recognizing it.

3.2 Higher-order terms and TRSs

The given definition of TRSs does not allow for "partial application" of a symbol, which is required to model higher-order programs where partially applied functions are first-class citizens. We overcome this limitation using *applicative TRS*, a construction first proposed by [Reynolds 1969] and studied thoroughly by [Kennaway et al. 1996], where partial applications are encoded using a special binary symbol @ added to Σ where $@(f, x)$ encode the application of f on x. For the sake of readability, in the rest of this paper we omit the use of @, and directly write $filter\ f\ l$ instead of $@(@(filter, f), l)$. Note that underneath, first-order term rewriting systems are still used.

3.3 Functional TRS

This work focuses on a family of term rewriting systems that we call "Functional Term Rewriting Systems" where each rule respects the following limitations, found in most higher-order functional languages: (1) each rule is *left-linear*, a variable cannot occur twice on the left hand side; (2) the left hand side cannot be just a variable, rules of the form $x \rightarrow t$ are forbidden; (3) subterms of a left hand side are irreducible, it is a pattern matching on values; (4) the alphabet Σ can be split into two sets C and $\mathcal{F}$ where C is a set of *constructor symbols* and $\mathcal{F}$ a set of *function symbols* (or defined symbols) such that $\mathcal{F}$ contains all the root symbols of the left-hand sides of all rewrite rules. For functional TRSs, the set of ground normal forms *GNF*($\mathcal{R}$) is also called the set of *values*. Values are either ground constructor terms or partial applications. Ground terms built with symbols of C are called *constructor terms*. The set of constructor terms is $\mathcal{T}(C)$. *Partial applications* are ground normal forms containing the @ symbol, e.g., $@(filter, even)$ where $@ \in \mathcal{F}$, and *filter*, *even* $\in C$.

4 REGULAR ABSTRACT INTERPRETATION

A standard approach for verifying properties of programs manipulating unbounded data structures consists in finding a finite abstraction (or at least an abstraction with a finite representation) that over-approximates the behavior of the program, and then check the correctness of the property on the resulting finite abstract model. In this section we define a regular abstract interpretation framework that adapts classical abstract interpretation to term rewriting systems and regular languages. In this framework, terms representing states of the execution are abstracted into a regular language, denoted by a state of a tree automaton. For readability, we denote those states by identifiers of the form *name*$^{\#}$, where *name* is an arbitrary symbol whose role is to provide an intuition of the language recognized by *name*$^{\#}$. In the following, we also call those states *abstract values*. The set of abstract value is the *abstract domain*, denoted by $\Sigma^{\#}$. The semantics of the program $\mathcal{R}$ is represented by an abstract TRS $\mathcal{R}^{\#}$ that operates over the abstract domain of regular languages.

We start with an example. Define I as the set $\{nil, cons(a, nil), cons(b, nil), ...\}$ of (not nested) lists of as and bs and assume that the safety property we wish to verify is

$$(1) \quad \forall l \in I.\ sorted\ (sort\ l) \not\rightarrow_{\mathcal{R}}^{*} false$$

This can be translated into an abstract interpretation problem where the goal is to find an *abstract* domain $\Sigma^{\#}$ provided with an abstraction function $\alpha : \mathcal{T}(\Sigma) \rightarrow \mathcal{P}(\Sigma^{\#})$ and a concretization function $\gamma : \Sigma^{\#} \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$, and an abstract semantics $\rightarrow^{\#}$ extracted from $\mathcal{R}$ which faithfully models $\mathcal{R}$ such that

$$(2) \quad sorted\ (sort\ ab\_list^{\#}) \not\rightarrow^{\#*} false^{\#}$$

where $ab\_list^\#$ and $false^\#$ are two abstract values of $\Sigma^\#$ such that $\alpha(I) = \{ab\_list^\#\}$ and $\gamma(false^\#) = \{false\}$. Note that having an abstraction α returning a set of abstract elements is non-standard in abstract interpretation but we have ensured that it is used consistently throughout the paper. In this particular first abstraction, all lists of a and b are abstracted by the same abstract element $ab\_list^\#$ thus $\alpha(I)$ is the singleton $\{ab\_list^\#\}$. Since $\Sigma^\#$ is finite, it is easier to verify property (2) than property (1). Then, if the abstraction is correct and property (2) is true, we can then deduce that property (1) is also true. In the rest of this section, we show how to define $\to^\#$, α and γ in terms of term rewriting systems and how to infer them with an abstract domain $\Sigma^\#$ which together will allow us to verify a given property. In the rest of this paper, we use σ to denote *concrete* substitutions from $\mathcal{X}$ to $\mathcal{T}(\Sigma)$, and π to denote *abstract* substitutions from $\mathcal{X}$ to $\Sigma^\#$. In addition we name $\gamma(\pi)$ the set of all concrete substitutions extracted from π, i.e., $\gamma(\pi) = \{\ \sigma \mid \forall x \in Dom(\pi).\ \sigma(x) \in \gamma(\pi(x))\ \}$.

4.1 Regular abstract domain

The *concrete domain* of a regular abstract interpretation is $\mathcal{T}(\Sigma)$. The elements of the *abstract domain* of a regular abstract interpretation are arbitrary symbols, each of them representing a regular set of terms of $\mathcal{T}(\Sigma)$. One originality of our approach resides in representing an abstract domain by a tree automaton, where each state corresponds to a regular set of terms. More precisely, an abstraction is represented by a tree automaton $\Lambda = \langle \Sigma, \Sigma^\#, \Sigma^\#, \Delta^\# \rangle$ where $\Sigma^\#$ is the set of abstract values and $\Delta^\#$ a term rewriting system defining the associated abstraction function α that describes how concrete terms are abstracted into abstract values. For readability, we write $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$. In the previous example, Λ can be defined using the following $\Delta^\#$:

$$\begin{aligned} a &\to ab^\# & nil &\to ab\_list^\# \\ b &\to ab^\# & cons(ab^\#, ab\_list^\#) &\to ab\_list^\# \end{aligned}$$

Note that a configuration of Λ is a term of $T(\Sigma, \Sigma^\#)$, a mix between abstract and concrete terms which allows the transition from the concrete domain to the abstract one. Each term of $T(\Sigma, \Sigma^\#)$ is called an *abstract pattern.*

Definition 4.1 (Regular Abstract Domain). Let Σ be an alphabet. A $T(\Sigma)$-abstraction $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ is a *regular abstract domain* iff the corresponding tree automaton $\langle \Sigma, \Sigma^\#, \Sigma^\#, \Delta^\# \rangle$ is normalized and complete. The abstraction and concretization functions are then defined by

$$\alpha(t) = \{a^\# \mid t \to^*_{\Delta^\#} a^\#\} \qquad \gamma(a^\#) = \{t \mid t \to^*_{\Delta^\#} a^\#\}$$

Note that in this definition α returns the set of all possible abstract values for any term t. Since Λ can be non-deterministic, this set may contain more than one abstract element.

4.2 Abstract semantics

By rewriting a term t using $\Delta^\#$, we can abstract some subterms of t by elements of $\Sigma^\#$. However, this abstracted term can no longer be rewritten using $\mathcal{R}$. We need to introduce a new rewriting system $\mathcal{R}^\#$ as an abstraction of $\mathcal{R}$, rewriting *abstract patterns* while *preserving the behavior* of $\mathcal{R}$. To deduce a property of $\mathcal{R}$ using $\mathcal{R}^\#$, the abstract $\mathcal{R}^\#$ must itself respect specific constraints w.r.t. $\mathcal{R}$.

Definition 4.2 (Abstraction of a TRS). Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ a regular abstract domain over $GNF(\mathcal{R})$. Let $\mathcal{R}^\#$ be a TRS over $T(\Sigma^\#)$. $\mathcal{R}^\#$ is an *abstraction* of $\mathcal{R}$ over Λ iff

- every rule of $\mathcal{R}^\#$ has the form $f(a^\#_1, \ldots, a^\#_n) \to a^\#$ with $f \in \Sigma^n$, $a^\#_1, \ldots, a^\#_n, a^\# \in \Sigma^\#$;
- (soundness) for all rules $f(a^\#_1, \ldots, a^\#_n) \to a^\#$ of $\mathcal{R}^\#$, for all terms t of $\{f(t_1, \ldots, t_n) \mid t_i \in \gamma(a^\#_i)\}$, for all terms $u \in GNF(\mathcal{R})$ such that $t \to^*_{\mathcal{R}} u$ then $u \in \gamma(a^\#)$;

- (completeness) for all symbols f used in $\mathcal{R}^\#$, for all terms $t = f(t_1, \ldots, t_n)$ where $t_i \in \mathit{GNF}(\mathcal{R})$ for all i, there must exist some rule $f(a_1^\#, \ldots, a_n^\#) \rightarrow a^\#$ such that for each i, $t_i \in \gamma(a^\#)$. In other words, for a given used symbol, each possible abstract input is mapped to at least one abstract output.

Theorem 4.3 (Correctness). *Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstract domain over $\mathit{GNF}(\mathcal{R})$. If $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$ over Λ then we have for all pattern p, abstract substitution π and abstract value $v^\# \in \Sigma^\#$:*

$$p\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} u^\# \quad \Rightarrow \quad \forall \sigma \in \gamma(\pi).\ \forall u \in \mathit{GNF}(\mathcal{R}).\ p\sigma \rightarrow^*_{\mathcal{R}} u \Rightarrow u \in \gamma(u^\#)$$

with $\gamma(\pi) = \{\ \sigma \mid \forall x \in \mathit{Dom}(\pi).\ \sigma(x) \in \gamma(\pi(x))\ \}$ the set of all possible concretized substitutions.

Proof. This can be proved by a simple induction on the size of the (abstract) rewriting path using the *soundness* property of $\mathcal{R}^\#$. If $p\pi = u^\#$ then p is some variable x. For all $\sigma \in \gamma(\pi)$, $p\sigma$ is a ground term. So for all $u \in \mathit{GNF}(\mathcal{R})$ such that $p\sigma \rightarrow^*_{\mathcal{R}} u$ we have $u = p\sigma$, with $u \in \gamma(u^\#)$. Next, if $p\pi \rightarrow^{k+1}_{\mathcal{R}^\# \cup \Delta^\#} u^\#$. By definition (of $\mathcal{R}^\#$ and $\Sigma^\#$), there exists a context C, a pattern $l = f(l_1, \ldots, l_n)$ and an abstract value $r^\#$ such that $p = C[l]$ and $p\pi \rightarrow_{\mathcal{R}^\# \cup \Delta^\#} C[r^\#] \rightarrow^k_{\mathcal{R}^\# \cup \Delta^\#} u^\#$. Now let us consider $\sigma \in \gamma(\pi)$ and $u \in \mathit{GNF}(\mathcal{R})$. Since $C[l]\pi \rightarrow_{\mathcal{R}^\# \cup \Delta^\#} C[r^\#]$ then by definition of $\mathcal{R}^\#$ (soundness of abstraction $\mathcal{R}^\#$) there exists some term r such that $l\sigma \rightarrow^*_{\mathcal{R}} r$ so that $C[l]\sigma \rightarrow^*_{\mathcal{R}} C[r]$ which we can write as (1) $p\sigma \rightarrow^*_{\mathcal{R}} C[r]\sigma$. Finally let x be a fresh variable and π' be the abstract substitution such that $\pi' = \pi \cup \{x \mapsto r^\#\}$. By construction we have $C[r^\#]\pi = C[x]\pi'$. Using the induction hypothesis on $C[x]\pi' \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} u^\#$, we get that (2) $\forall \sigma' \in \gamma(\pi').\ \forall u \in \mathit{GNF}(\mathcal{R}).\ C[x]\sigma' \rightarrow^*_{\mathcal{R}} u \Rightarrow u \in \gamma(u^\#)$. Since $\pi' = \pi \cup \{x \mapsto r^\#\}$, for all $\sigma' \in \gamma(\pi')$ there exists $r \in \gamma(r^\#)$ and $\sigma \in \gamma(\pi)$ such that $\sigma' = \sigma \cup \{x \mapsto r\}$. In this case, $C[x]\sigma' = C[r]\sigma$ and we can connect (1) and (2) to obtain that $p\sigma \rightarrow^*_{\mathcal{R}} u \Rightarrow u \in \gamma(u^\#)$. □

Note that it is not sufficient to have *some* abstraction of $\mathcal{R}$ to be able to verify the desired property. On our previous example, the following TRS is a correct abstraction of $\mathcal{R}$:

$$\mathit{sort}\ \mathit{ab\_list}^\# \rightarrow \mathit{ab\_list}^\# \qquad\qquad \mathit{sorted}\ \mathit{ab\_list}^\# \rightarrow \mathit{bool}^\#$$

however since all lists of *a*s and *b*s are always abstracted by the same element *ab_list*$^\#$, this abstraction is too coarse to prove that for all list l, $\mathit{sorted}\ (\mathit{sort}\ l) \not\rightarrow^*_{\mathcal{R}} \mathit{false}$. To succeed, we need to make sure that our abstraction provides the additional property:

$$\forall v^\#.\ \mathit{sorted}\ (\mathit{sort}\ v^\#) \not\rightarrow^*_{\mathcal{R}\# \cup \Lambda} \mathit{false}^\# \Rightarrow \forall l.\ \mathit{sorted}\ (\mathit{sort}\ l) \not\rightarrow^*_{\mathcal{R}} \mathit{false}$$

Another way to phrase it is for $\mathcal{R}^\#$ to be "complete" w.r.t $\mathit{sorted}\ (\mathit{sort}\ l)$ and $\mathit{false}^\#$, according to the following definition:

Definition 4.4 (Complete abstraction). Let $\mathcal{R}^\#$ be a TRS abstraction of $\mathcal{R}$ over $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$. We say that $\mathcal{R}^\#$ is complete with regard to a pattern p and abstract value $v^\#$ when for all term $t \in \gamma(v^\#)$, for all substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\Sigma)$, if $p\sigma \rightarrow^*_{\mathcal{R}} t$ then there exists an abstract substitution $\pi : \mathcal{X} \mapsto \Sigma^\#$ such that $\sigma \in \gamma(\pi)$ and $p\pi \rightarrow^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$. This can be seen as the the contraposition of Theorem 4.3 for a particular case of p and $v^\#$.

In our example, for $\mathcal{R}^\#$ to be complete w.r.t $\mathit{sorted}\ (\mathit{sort}\ l)$ and *false*$^\#$ we need at least three elements to abstract lists of *a*s and *b*s: *sorted*$^\#$, *unsorted*$^\#$, and *b_list*$^\#$ recognizing respectively sorted a and

b lists, unsorted lists, and lists of bs. The abstract domain Λ becomes:

$$
\begin{array}{rcc}
a \rightarrow a^{\#} & cons(b^{\#}, b\_list^{\#}) \rightarrow b\_list^{\#} & cons(b^{\#}, sorted^{\#}) \rightarrow unsorted^{\#} \\
b \rightarrow b^{\#} & cons(a^{\#}, b\_list^{\#}) \rightarrow sorted^{\#} & cons(b^{\#}, unsorted^{\#}) \rightarrow unsorted^{\#} \\
nil \rightarrow b\_list^{\#} & cons(a^{\#}, sorted^{\#}) \rightarrow sorted^{\#} & cons(a^{\#}, unsorted^{\#}) \rightarrow unsorted^{\#}
\end{array}
$$

and the abstract TRS $\mathcal{R}^{\#}$ becomes:

$$
\begin{array}{rcrc}
sort\ b\_list^{\#} \longrightarrow b\_list^{\#} & & sorted\ b\_list^{\#} \longrightarrow true^{\#} \\
sort\ sorted^{\#} \longrightarrow sorted^{\#} & & sorted\ sorted^{\#} \longrightarrow true^{\#} \\
sort\ unsorted^{\#} \longrightarrow sorted^{\#} & & sorted\ unsorted^{\#} \longrightarrow false^{\#}
\end{array}
$$

Using those abstract elements, domain and TRS, we can show:

$$
\left.\begin{array}{l}
sorted\ (sort\ sorted^{\#}) \not\rightarrow^{*}_{\mathcal{R}^{\#}\cup\Lambda} false^{\#} \\
sorted\ (sort\ unsorted^{\#}) \not\rightarrow^{*}_{\mathcal{R}^{\#}\cup\Lambda} false^{\#} \\
sorted\ (sort\ b\_list^{\#}) \not\rightarrow^{*}_{\mathcal{R}^{\#}\cup\Lambda} false^{\#}
\end{array}\right\} \quad \Rightarrow \quad \forall l \in I.\ sorted\ (sort\ l) \not\rightarrow^{*}_{\mathcal{R}} false
$$

To use this for verification, we need to *infer* Λ with an $\mathcal{R}^{\#}$ that is *complete* with regard to our desired property: this is the *inference problem* we solve in this paper.

Definition 4.5 (Inference problem). Assume that we are given a TRS $\mathcal{R}$, a pattern $p \in T(\Sigma, \mathcal{X})$, an initial abstract domain $\Lambda_* = \langle \Sigma^{\#}_*, \Delta^{\#}_* \rangle$ and a target abstract value $v^{\#} \in \Sigma^{\#}_*$. A solution to the inference problem is

(1) an abstract domain $\Lambda = \langle \Sigma^{\#}, \Delta^{\#} \rangle$ such that $\Sigma^{\#} \supseteq \Sigma^{\#}_*$ and $\Delta^{\#} \supseteq \Delta^{\#}_*$;
(2) an abstraction $\mathcal{R}^{\#}$ of $\mathcal{R}$ in Λ, complete w.r.t. p and $v^{\#}$.
(3) the set Π of all the abstract substitutions π such that $p\pi \longrightarrow^{*}_{\Delta^{\#}\cup\mathcal{R}^{\#}} v^{\#}$.

Intuitively, a solution provides a set of substitutions from variables to abstract values such that if the pattern rewrites to a result that belongs to $v^{\#}$ then there is an abstract substitution in Π such that the pattern rewrites to $v^{\#}$. Note that if the resulting set Π is empty, we can deduce that for all $t \in \gamma(v^{\#})$, for all substitution σ we have $p\sigma \not\rightarrow^{*}_{\mathcal{R}} t$.

4.3 Building the abstraction using a CEGAR procedure

We chose to find a correct abstraction for our property using a Counter Example Guided Abstraction Refinement (CEGAR) procedure [Clarke et al. 2000]. We start the procedure with initial abstractions $\mathcal{R}^{\#}$ and Λ, and look for counter-examples to refine them. For instance, assume that we start from Λ as it is defined in Section 4.1.

$$
a \rightarrow ab^{\#} \qquad nil \rightarrow ab\_list^{\#} \qquad b \rightarrow ab^{\#} \qquad cons(ab^{\#}, ab\_list^{\#}) \rightarrow ab\_list^{\#}
$$

On this abstract domain, $\mathcal{R}^{\#}$ has rules: $sort\ ab\_list^{\#} \rightarrow ab\_list^{\#}$ and $sorted\ ab\_list^{\#} \rightarrow bool^{\#}$, where $\gamma(bool^{\#}) = \{true, false\}$. Obviously, this abstraction is too coarse because it does not distinguish between *true* and *false*. As a result, $sorted\ (sort\ ab\_list^{\#}) \rightarrow^{*}_{\mathcal{R}^{\#}\cup\Lambda} bool^{\#}$ where $\gamma(bool^{\#})$ contains $false$, i.e., the forbidden term. As a result, this abstraction cannot be used to prove that $\forall l \in I.\ sorted\ (sort\ l) \not\rightarrow^{*}_{\mathcal{R}} false$. A natural way to refine this abstraction is to separate $ab\_list^{\#}$ into several classes such that the abstraction of *sorted* distinguishes between *true* and *false*, i.e., find two lists l_1 and l_2 such that $\alpha(l_1) = \alpha(l_2) = \{ab\_list^{\#}\}$ and $(sorted\ l_1) \rightarrow^{*}_{\mathcal{R}} true$ and $(sorted\ l_2) \rightarrow^{*}_{\mathcal{R}} false$. A naive way to find such counter-example is to enumerate lists abstracted by $ab\_list^{\#}$ until we find two lists for which the *sorted* function answers *true* for the first and *false* for the second.

$$\begin{array}{rr} sorted\ nil \to^*_{\mathcal{R}} true & sorted\ cons(a, nil) \to^*_{\mathcal{R}} true \\ sorted\ cons(b, nil) \to^*_{\mathcal{R}} true & sorted\ cons(a, cons(b, nil)) \to^*_{\mathcal{R}} true \\ sorted\ cons(b, cons(a, nil)) \to^*_{\mathcal{R}} false & \end{array}$$

Here, we can choose either $l_1 = nil$, $l_1 = cons(a, nil)$, $l_1 = cons(b, nil)$, or $l_1 = cons(a, cons(b, nil)))$ and $l_2 = cons(b, cons(a, nil))$. In the CEGAR framework, l_1 with $\alpha(l_1) = \{ab\_list^\#\}$ is called a *spurious counter-example*, i.e., an abstract derivation ($sorted\ ab\_list^\# \to bool^\#$ where $\gamma(bool^\#)$ contains $false$) which has no concrete counterpart ($sorted\ l_1 \to^*_{\mathcal{R}} true$). Note that l_2 is also a spurious counter-example because $\alpha(l_2) = \{ab\_list^\#\}$, $sorted\ ab\_list^\# \to bool^\#$, $\gamma(bool^\#)$ contains $true$, and $sorted\ l_2 \to^*_{\mathcal{R}} false$. The abstraction refinement then consists in defining a new abstract domain separating the lists $l_1 = nil$, $l_1 = cons(a, nil)$, …, from the list $l_2 = cons(b, cons(a, nil))$. With this refined abstract domain, we compute the new $\mathcal{R}^\#$, and iterate this process until we find a real counter-example (*i.e.*, a list l such that $sorted\ l$ rewrites to *false*, the forbidden term) or the property is proven (no spurious counter-example has to be refined).

4.4 Three challenges to tackle for using CEGAR in practice

To build a program verification technique from this we have to solve three problems. First, building a new abstract domain by separating only one concrete value at each refinement step may lead to non-termination. For instance, in our example, we may refine to separate $cons(b, cons(a, nil))$ from the rest, then separate $cons(a, cons(b, cons(a, nil)))$, etc. This is used for instance in [Ong and Ramsay 2011]. However, there are ways to explore the set of possible abstractions so that the refinement procedure is guaranteed to terminate if there exists a regular abstraction $\mathcal{R}^\#$ and Λ proving or disproving the property. This exploration technique generally uses SMT-solvers to explore the set of possible abstractions $\Sigma^\#$ w.r.t. their cardinal [Matsumoto et al. 2015]. The second problem is that, given an abstract domain, finding spurious counter-examples is not easy. Indeed, the TRS $\mathcal{R}^\#$ does not explicitly encode the rewriting relation between terms. As a consequence, even if we know that a (forbidden) term may be reachable, the rewriting paths to this term must be recalculated afterward to give a complete counter-example. This is computationally expensive. This is illustrated above where it took five rewritings to find one counter-example, i.e., lists l_1 and l_2. However, in the above example, the rewritings only depend on a single function: $sorted$. In practice, to find counter-examples on complex programs it is necessary to rewrite using several function definitions. Thus, the computational cost for finding a counter-example grows with the size of the program to verify. Finally, the third problem is that the "abstract and refine" procedure presented above is not modular. From a set of constraints $l_1 \neq l_2$ that we extract from a spurious counter-example, the next abstraction is recomputed for the *whole* program. As a result the size of the program directly and greatly impacts the efficiency of two main steps of the procedure: the search for counter-examples and the abstraction refinement.

This is why we want to define a *modular* procedure able to analyse functions independently and whose *termination* is guaranteed if there exists a regular abstraction satisfying the property. To ensure termination, as we will see in Section 5.4, we also use an SMT-based technique exploring possible abstraction w.r.t. the size of the abstract domain $\Sigma^\#$. For modularity, we use a *type system* attaching abstraction information to each function (to each symbol of the TRS). In the next section, we show how to translate the above inference problem into a *type inference problem* over regular language types, that will allow us to design a modular inference procedure for those types.

5 REGULAR LANGUAGE TYPES

A convenient way of modularizing the abstract interpretation is by introducing a type system to attach abstraction information to each term and symbol. In this approach, the *set of types* is the abstract set of values $\Sigma^\#$ and a term has type τ if the term rewrites to τ using $\Delta^\#$ and $\mathcal{R}^\#$. In practice, each type represents a regular language. A *type substitution* $\pi \in \mathcal{X} \to \Sigma^\#$ maps variables to types. We say that the abstract semantics $\mathcal{R}^\#$ is the *type environment* of symbols. In the following, we define a typing judgment $\vdash$ which can be used to give types to patterns, relative to a given $\Lambda, \mathcal{R}^\#$ and substitution π which assigns a type to each variable of the pattern. The typing judgment $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$ means that the pattern $p \in T(\Sigma, \mathcal{X})$ can be typed with $\tau \in \Sigma^\#$ using the substitution π, Λ and $\mathcal{R}^\#$.

Definition 5.1 (Typing rules). Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ be an abstraction of $T(\Sigma)$, $\mathcal{R}^\#$ a *type environment* and π a substitution from $\mathcal{X}$ to $\Sigma^\#$. We define the typing judgment $\vdash$ via the following inference rules. In the rules we use the rewriting rules $p \to \tau$ of $\Delta^\#$ to deduce a type for constructor and function applications. Recall that Σ is the disjoint union of constructor symbols C and function symbols $\mathcal{F}$.

$$\text{var}\,\frac{\pi(x) = \tau}{\Lambda, \mathcal{R}^\#, \pi \vdash x : \tau}$$

$$\text{constructor}\,\frac{\Lambda, \mathcal{R}^\#, \pi \vdash p_1 : \tau_1 \quad \dots \quad \Lambda, \mathcal{R}^\#, \pi \vdash p_n : \tau_n \quad f \in C \quad f(\tau_1, \dots, \tau_n) \to \tau \in \Delta^\#}{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau}$$

$$\text{sub-typing}\,\frac{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau' \quad \tau' \to \tau \in \Delta^\#}{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau}$$

$$\text{application}\,\frac{\Lambda, \mathcal{R}^\#, \pi \vdash p_1 : \tau_1 \quad \dots \quad \Lambda, \mathcal{R}^\#, \pi \vdash p_n : \tau_n \quad f \in \mathcal{F} \quad f(\tau_1, \dots, \tau_n) \to \tau \in \mathcal{R}^\#}{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau}$$

The *var* rule uses the type substitution π to type a single variable. The *application* rule follows the abstract semantics $\mathcal{R}^\#$ to type a pattern which can be rewritten. The *constructor* rule uses the abstraction Λ, and in particular $\Delta^\#$, to give a type to patterns built from a constructor symbol. The *sub-typing* rule does the same using the ϵ-transitions of $\Delta^\#$. From this rule we extract a sub-typing relation $\preceq$ where $\tau_1 \preceq \tau_2$ means that $\tau_1 \to^*_{\Delta^\#} \tau_2$.

The typing judgment definition makes a bridge between the rewriting world and the typing world. Each typing rule correspond to a rewriting step with either $\mathcal{R}^\#$ which becomes the *type environment*, or $\Delta^\#$ which includes the *types definitions*. The following lemma states the correctness of the type system using the abstract interpretation defined in the previous section. Its proof uses the rewrite system $\mathcal{R}^\# \cup \Delta^\#$ as an intermediate step between type system and abstract interpretation.

Theorem 5.2 (Correctness). *Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstraction of $T(\Sigma)$ and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ over Λ. For all patterns p and types τ (which are abstract values of $\Sigma^\#$) we have:*

$$\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau \iff p\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$$

Proof. First, let's show that $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$ implies $p\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$. We show this with a simple induction on the type inference rules.

- (var) $\Lambda, \mathcal{R}^\#, \pi \vdash x : \tau$. Then $\pi(x) = \tau$, $p = x$ and thus $p\pi = \tau$.

- (constructor) $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \ldots, p_n) : \tau$ with $f \in C$. Then $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau_i$. By induction hypothesis this means that $p_i\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$. In addition, by definition $f(\tau_1, \ldots, \tau_n) \rightarrow \tau \in \Delta^\#$. We conclude that $f(p_1, \ldots, p_n)\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.
- (sub-typing) $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \ldots, p_n) : \tau$ with $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \ldots, p_n) : \tau'$ and $\tau' \rightarrow \tau \in \Delta^\#$. By induction hypothesis we have that $f(p_1, \ldots, p_n)\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau'$. Using $\tau' \rightarrow \tau \in \Delta^\#$ we conclude that $f(p_1, \ldots, p_n)\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.
- (application) $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \ldots, p_n) : \tau$ with $f \in \mathcal{F}$. Then $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau_i$. By induction hypothesis this means that $p_i\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$. In addition, by definition $f(\tau_1, \ldots, \tau_n) \rightarrow \tau \in \mathcal{R}^\#$. We conclude that $f(p_1, \ldots, p_n)\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.

First, let's show that $p\pi \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$ implies $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$. We use a strong induction on the length of the rewriting path. First if $p\pi = \tau$ then p is a variable x and we have $\pi(x) = \tau$. Because of the inference rule (var) we have $\Lambda, \mathcal{R}^\#, \pi \vdash x : \tau$. Now if $p\pi \rightarrow^{k+1}_{\mathcal{R}^\# \cup \Delta^\#} \tau$, we need to consider the following cases:

- $p\pi \rightarrow^k_{\mathcal{R}^\# \cup \Delta^\#} f(\tau_1, \ldots, \tau_n) \rightarrow_{\Delta^\#} \tau$ with $f \in C$ and $p = f(p_1, \ldots, p_n)$. For all $i = 1 \ldots n$, we have $p_i\pi \rightarrow^{k_i}_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$ with $k_i \leq k$. We can apply the induction hypothesis on all of them and get $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau$ for $i = 1 \ldots n$. Then, using the *constructor* type inference rule gives us $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$.
- $p\pi \rightarrow^k_{\mathcal{R}^\# \cup \Delta^\#} \tau' \rightarrow_{\Delta^\#} \tau$. Using the induction hypothesis on $p\pi \rightarrow^k_{\mathcal{R}^\# \cup \Delta^\#} \tau'$ we get that $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau'$. We can then apply the *sub-typing* type inference rule to get $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$.
- $p\pi \rightarrow^k_{\mathcal{R}^\# \cup \Delta^\#} f(\tau_1, \ldots, \tau_n) \rightarrow_{\mathcal{R}^\#} \tau$ with $f \in \mathcal{F}$ and $p = f(p_1, \ldots, p_n)$. For all $i = 1 \ldots n$, we have $p_i\pi \rightarrow^{k_i}_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$ with $k_i \leq k$. Thus, we can apply the induction hypothesis and get $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau$ for $i = 1 \ldots n$. We can then apply the *application* type inference rule to get $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$.

□

REMARK 5.1. *Recall that we encode higher-order using the dedicated @ symbol (Section 3.2). For instance $@(@(f, x), y)$ is the total application of f on two parameters x and y. A reader familiarized with usual ML-like type systems may notice the lack of arrow type for partial applications such as $@(f, x)$. We do not need them in our type system as they can be represented using regular languages, just like any other type. The ML-type $\tau_1 \rightarrow \tau_2$ is represented by the regular language of all terms $t \in \mathcal{T}(\Sigma)$ such that the application $@(t, x)$ rewrites to a term of τ_2 when x rewrites to a term of τ_1.*

Example 5.3. Consider $\mathcal{R}$ defining a (buggy) *delete* function as

$$\begin{array}{ll} (d1) & delete\ x\ nil \rightarrow nil \\ (d2) & delete\ x\ cons(y, tail) \rightarrow if(eq(x, y), delete\ x\ tail, delete\ x\ tail) \end{array}$$

The definition of the equality predicate *eq* and of the if-then-else symbol *if* are omitted but present in $\mathcal{R}$. The *delete* function is supposed to remove every occurrence of x in the given list. In the last rule however, we forgot to put y back in the list when $x \neq y$. As a result, this *delete* function always return *nil*. This can be spotted by noticing that there exists an abstraction Λ and $\mathcal{R}^\#$ such that $\Lambda, \mathcal{R}^\#, \pi \vdash delete\ x\ l\ : nil^\#$ with $\pi(x) = ab^\#$ and $\pi(l) = ab\_list^\#$. Let the abstraction Λ of $T(\Sigma)$ be defined by

$$\begin{array}{llll} a \rightarrow ab^\# & true \rightarrow bool^\# & nil \rightarrow ab\_list^\# & nil \rightarrow nil^\# \\ b \rightarrow ab^\# & false \rightarrow bool^\# & cons(ab^\#, ab\_list^\#) \rightarrow ab\_list^\# & nil^\# \rightarrow ab\_list^\# \end{array}$$

and let the abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ over Λ be defined by

$$eq(ab^\#,\ ab^\#) \longrightarrow bool^\# \qquad if(bool^\#,\ nil^\#,\ nil^\#) \longrightarrow nil^\# \qquad delete\ ab^\#\ ab\_list^\# \longrightarrow nil^\#$$

It is then easy to show $\Lambda, \mathcal{R}^\#, \pi \vdash delete\ x\ l\ : nil^\#$ by the following typing derivation ($\Lambda, \mathcal{R}^\#, \pi$ are omitted to improve the readability):

$$\dfrac{\dfrac{\pi(x) = ab^\#}{\vdash x : ab^\#} \qquad \dfrac{\pi(l) = ab\_list^\#}{\vdash l : ab\_list^\#} \qquad delete\ ab^\#\ ab\_list^\# \longrightarrow nil^\# \in \mathcal{R}^\#}{\vdash delete\ x\ l : nil^\#}$$

It is possible to build an abstraction $\mathcal{R}^\#$ from Λ and $\mathcal{R}$ so as to give the most precise regular type information to each function. The outline of the procedure is as follows: For each function f of $\mathcal{R}$, for every combination of input and output types, make the hypothesis that this combination is valid and try to type both sides of each rule defining f in $\mathcal{R}$ with this combination of types. If it is possible, then the combination is valid. From the set of valid combinations, keep the most precise one.

Example 5.4. In the previous example, the combination $delete\ ab^\#\ ab\_list^\# \longrightarrow bool^\#$ is not valid for the *delete* function since it is impossible to type the right-hand-side of the rule $(d1)$ of *delete* with $bool^\#$. The combination $delete\ ab^\#\ ab\_list^\# \longrightarrow nil^\#$ that has been selected is valid. The right-hand side of the rule $(d1)$ of *delete* has the type $nil^\#$ by the typing derivation:

$$\dfrac{nil \longrightarrow nil^\# \in \Delta^\#}{\vdash nil\ \ : nil^\#}$$

The typing judgment of the right-hand-side of rule $(d2)$ with $nil^\#$ comes from the following derivation (where we omit the rules of *if*, *eq* and *delete* in $\mathcal{R}^\#$ for readability):

$$\dfrac{\dfrac{\dfrac{\pi(x) = ab^\#}{\vdash x : ab^\#} \qquad \dfrac{\pi(y) = ab^\#}{\vdash y : ab^\#}}{\vdash eq(x, y) : bool^\#} \qquad \dfrac{\dfrac{\pi(x) = ab^\#}{\vdash x : ab^\#} \qquad \dfrac{\pi(tail) = ab\_list^\#}{\vdash tail : ab\_list^\#}}{\vdash delete\ x\ tail : nil^\#}}{\vdash if(eq(x, y),\ delete\ x\ tail,\ delete\ x\ tail) : nil^\#}$$

Note that the combination $delete\ ab^\#\ ab\_list^\# \longrightarrow ab\_list^\#$ is also valid for the *delete* function, but is less precise than $delete\ ab^\#\ ab\_list^\# \longrightarrow nil^\#$.

This example illustrates that using this type system and a given abstraction Λ of $T(\Sigma)$, there exists a procedure to build an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ from Λ that gives precise information about the functions of the program. In this example, it is convenient to have $nil^\#$ as part of Λ, allowing us to type the *delete* function with $nil^\#$ as output, which in turns allowed us to spot the mistake in *delete* (cf. Section 6). In general, Λ does not contain enough information to prove or disprove the wanted property. The abstraction Λ must be *refined* along with $\mathcal{R}^\#$. This transforms the *abstraction inference problem* introduced in the previous section into the following *type inference problem.*

Definition 5.5 (Regular language type inference problem). Let $\mathcal{R}$ be a term rewriting system and p a pattern of $\mathcal{T}(\Sigma, \mathcal{X})$. Let $\Lambda_* = \langle \Sigma_*^\#, \Delta_*^\# \rangle$ be an initial abstract domain, and $\tau \in \Sigma_*^\#$ a (target) type. A solution to the type inference problem is (1) an abstract domain $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ such that $\Sigma^\# \supseteq \Sigma_*^\#$ and $\Delta^\# \supseteq \Delta_*^\#$; (2) an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in Λ, complete w.r.t. p and $v^\#$. (3) the set Π of all the type environments π such that $\Lambda, \mathcal{R}^\#, \pi \vdash p : v^\#$.

Note that type inference problems are usually concerned with finding *some* type substitution π such that $\pi \vdash p : \tau$. In our case however, since we want to capture the entire behavior of the input pattern with regard to the target type τ, we are interested in finding an abstraction containing *all* such type substitutions, in Π. For instance if we consider the pattern $p = xor(x, y)$ with the target type $\tau = true^\#$, a solution to the regular language type inference problem must include $\Lambda, \mathcal{R}^\#$ with Π containing π_1, π_2 such that $\pi_1 = \{x \mapsto true^\#, y \mapsto false^\#\}$ and $\pi_2 = \{x \mapsto false^\#, y \mapsto true^\#\}$. These two substitutions are necessary (and sufficient) to capture the entire behavior of the *xor* function on its input with regard to the target type $true^\#$.

5.1 Type partitions

The type inference procedure we define in this paper is fundamentally an inductive inference procedure working on the structure of the given pattern. However having multiple possible type environments for a single pattern and target type is not convenient, since we would need to analyse every case and "split" the analysis at each induction step.

Example 5.6. We want to type the term $xor(f(X), Y)$ with target type $true^\#$. The only possible abstraction for *xor* separating *true* and *false* is the following $\mathcal{R}^\#$:

$$\begin{array}{cc} xor(true^\#, true^\#) \to false^\# & xor(false^\#, true^\#) \to true^\# \\ xor(true^\#, false^\#) \to true^\# & xor(false^\#, false^\#) \to false^\# \end{array}$$

This means that to have $xor(f(X), Y) : true^\#$, we may have either $f(X) : true^\#$ or $f(X) : false^\#$. We need to analyse both cases, splitting the analysis of f into two branches. Then, depending on the definition of f, we may need to split again each of the two branches to analyze X, etc. This can result into an exponential blow-up.

We would like to have one single environment to pass along. The fundamental idea to achieve this is to use *type partition* environments instead of using type environments. A *type partition* is a set of types which represent non-overlapping (regular) sets of values and which together cover the whole domain of values. From a given type environment π we can always construct a type partition environment where each variable x maps to $\{\pi(x), \overline{\pi(x)}\}$ (where $\overline{\pi(x)}$ is the complement of the language represented by $\pi(x)$). We call this the *partitioned domain* of x with respect to π. Note that other partitions are possible, e.g., by further dividing the complement of $\pi(x)$.

In general, type partitions form a semi-lattice w.r.t $\sqsubseteq$, where $T_1 \sqsubseteq T_2$ iff for all element $a_1^\#$ of T_1 there exists an element $a_2^\#$ of T_2 such that $\gamma(a_1^\#) \subseteq \gamma(a_2^\#)$. The greatest lower bound of two partitions T_1 and T_2 is the partition T with the fewest elements such that $T \sqsubseteq T_1$ and $T \sqsubseteq T_2$. Following tree automata usage, we call this the *product* $T_1 \otimes T_2$. A type partition environment is a mapping from variables to type partitions. For a set Π of type environments, we define the type partition environment $\widetilde{\Pi}(x)$ to be the environment where the *partitioned domain* of a variable x is the product between all the partitioned domains of x in every environment of Π.

Example 5.7. Let $\Sigma^\# = \{a^\#, b^\#, c^\#, ab^\#, bc^\#\}$. We assume that $\gamma(ab^\#) = \gamma(a^\#) \cup \gamma(b^\#)$ and $\gamma(bc^\#) = \gamma(b^\#) \cup \gamma(c^\#)$. Let x be a variable, $\pi_1 = \{x \mapsto a^\#\}$, $\pi_2 = \{x \mapsto c^\#\}$, and $\Pi = \{\pi_1, \pi_2\}$. The *partitioned domain* of x in π_1 is $\{a^\#, bc^\#\}$ since $\overline{a^\#} = bc^\#$. The *partitioned domain* of x in π_2 is $\{ab^\#, c^\#\}$. The *partitioned domain* of x in Π is $\widetilde{\Pi}(x) = \{a^\#, bc^\#\} \otimes \{ab^\#, c^\#\} = \{a^\#, b^\#, c^\#\}$.

To solve the regular language type inference problem of a pattern p for the target type τ, it is sufficient to compute $\widetilde{\Pi}$ instead of Π. Indeed from $\widetilde{\Pi}$ we can extract another set of type environments, $\Pi' = \{\ \pi \mid \forall x \in Var(p).\ \pi(x) \in \widetilde{\Pi}(x)\ \}$, for which by construction for all $\pi \in \Pi'$ we have either $\pi \vdash p : \tau$ or $\pi \vdash p : \overline{\tau}$. So Π' is a super-set of Π extracted from $\widetilde{\Pi}$. Hence in the rest of this paper, we shall focus on finding $\widetilde{\Pi}$ instead of Π. This is the *type partition inference problem.*

Definition 5.8 (Regular language type partition inference problem). Let $\mathcal{R}$ be a term rewriting system and p a pattern of $\mathcal{T}(\Sigma, \mathcal{X})$. Let $\Lambda_* = \langle \Sigma^\#_*, \Delta^\#_* \rangle$ be an initial abstract domain, and a target type partition $T \in \mathcal{P}(\Sigma^\#_*)$. A solution to the type partition inference problem is (1) an abstract domain $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ such that $\Sigma^\# \supseteq \Sigma^\#_*$ and $\Delta^\# \supseteq \Delta^\#_*$; (2) an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in Λ, complete w.r.t p and every $v^\# \in T$; (3) a type partition environment $\widetilde{\Pi}$, from which we can derive $\Pi = \{\, \pi \mid \forall x \in Var(p).\ \pi(x) \in \widetilde{\Pi}(x) \,\}$ containing all the substitutions π such that $\Lambda, \mathcal{R}^\#, \pi \vdash p : v^\#$ with $v^\# \in T$.

Example 5.9. We return to the previous *xor* example and type the term $xor(f(X), Y)$ with the target type partition $\{true^\#, false^\#\}$. Again, the only possible abstraction for *xor* in $\mathcal{R}^\#$ is to have

$$\begin{aligned} &xor(true^\#, true^\#) \to false^\# &\qquad& xor(false^\#, true^\#) \to true^\# \\ &xor(true^\#, false^\#) \to true^\# &\qquad& xor(false^\#, false^\#) \to false^\# \end{aligned}$$

But now we only have one single inductive case to type f, which is to have $f(X)$ with the type partition $\{true^\#, false^\#\}$. As an example, if we assume that f is the identity function, then the result of the analysis gives $\widetilde{\Pi} = \{X \mapsto \{true^\#, false^\#\}, Y \mapsto \{true^\#, false^\#\}\}$. The resulting Π includes the four possible combinations, which are $\pi_1 = \{X \mapsto true^\#, Y \mapsto true^\#\}$, $\pi_2 = \{X \mapsto true^\#, Y \mapsto false^\#\}$, $\pi_3 = \{X \mapsto false^\#, Y \mapsto true^\#\}$ and $\pi_4 = \{X \mapsto false^\#, Y \mapsto false^\#\}$.

The rest of this section defines an inference procedures for solving this type partition inference problem. We first present the general algorithm. We then present the invariant learning procedure which uses SMT-solving for minimizing the size of the abstraction automaton.

5.2 Inference algorithm

This section introduces the type inference algorithm for any TRS. The main function of this algorithm, *partition-inference* is in charge of solving the type partition inference problem, that is to find correct type partitions for every variable of a given pattern we wish to type with a given type partition. This algorithm uses the auxiliary functions *analyze-function* and *merge*. The role of the *merge* function will be detailed below. The role of *analyze-function* is to compute the *type partitions signature* of a given symbol f for a given output type partition T. This correspond to the input type partitions needed for every term $t_1, \ldots, t_n$ to type $f(t_1, \ldots, t_n)$ with the given output type partition T. For instance, the expected type partition signature of *xor* of Example 5.9 for the output partition $\{true^\#, false^\#\}$ is $(\{true^\#, false^\#\}, \{true^\#, false^\#\}) \to \{true^\#, false^\#\}$.

```
function partition-inference
    input  : A TRS $\mathcal{R}$, an abstraction $\Lambda_*$, pattern $p$, and type partition $T$.
    output : A solution $\Lambda$, $\mathcal{R}^\#$ and $\widetilde{\Pi}$ to the partition inference problem of $\mathcal{R}, \Lambda_*, p$ and $T$.
    match $p$ with
        when $x$ then
            return $\{\, x \mapsto T \,\}$
        when $f(p_1, \ldots, p_n)$ then
            let $\Lambda', \mathcal{R}'^\# = (T_1, \ldots, T_n) =$ analyze-function($\mathcal{R}, \Lambda_*, f, T$);
            foreach pattern $p_i$ do
                let $\Lambda_i, \mathcal{R}^\#_i, \widetilde{\Pi}_i =$ partition-inference($\mathcal{R}'^\#, \Lambda', p_i, T_i$);
            return merge($\Lambda_1, \ldots, \Lambda_n, \mathcal{R}^\#_1, \ldots, \mathcal{R}^\#_n, \widetilde{\Pi}_i, \ldots, \widetilde{\Pi}_n$)
```

Algorithm 1: Inference of type partitions

Definition 5.10 (Type partitions signature). Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstract domain, and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ defined over Λ. Let f be a symbol of Σ. Let $T, T_1, \dots, T_n$ be type partitions over $\Sigma^\#$. We say that $(T_1, \dots, T_n) \longrightarrow T$ is a *type partition signature* for f in $\Lambda, \mathcal{R}^\#$ if for all π, all patterns p_i, and $\tau_i \in T_i$, such that $\pi, \Lambda, \mathcal{R}^\# \vdash p_i : \tau_i$, for $i = 1 \dots n$, then there exists $\tau \in T$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash f(p_1, \dots, p_n) : \tau$.

The second auxiliary function *merge* deduces the final solution $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle, \mathcal{R}^\#, \widetilde{\Pi}$ of the type partition inference problem from all the $\Lambda_i = \langle \Sigma_i^\#, \Delta_i^\# \rangle, \mathcal{R}_i^\#$ found for all sub-patterns p_i. It is defined as the smallest sets such that (1) $\Sigma^\# \supseteq \Sigma_1^\# \cup \dots \cup \Sigma_n^\#$; (2) $\Delta^\# \supseteq \Delta_1^\# \cup \dots \cup \Delta_n^\#$; (3) $\mathcal{R}^\# = \mathcal{R}_*^\# \cup \mathcal{R}_1^\# \cup \dots \cup \mathcal{R}_n^\#$; (4) $\widetilde{\Pi} = \widetilde{\Pi}_1 \cup \dots \cup \widetilde{\Pi}_n$ where the union of two type partition environments $\widetilde{\Pi} = \widetilde{\Pi}_1 \cup \widetilde{\Pi}_2$ is defined for each variable of $Dom(\widetilde{\Pi}_1) \cup Dom(\widetilde{\Pi}_2)$ by

$$\widetilde{\Pi}(x) = \begin{cases} \widetilde{\Pi}_1(x) & \text{if} \quad x \notin Dom(\widetilde{\Pi}_2) \\ \widetilde{\Pi}_2(x) & \text{if} \quad x \notin Dom(\widetilde{\Pi}_1) \\ \widetilde{\Pi}_1(x) \otimes \widetilde{\Pi}_2(x) & \text{otherwise} \end{cases}$$

Hence, assuming that *analyze-function* is correct, we can then prove that the whole *partition-inference* algorithm is correct (cf. Appendix).This algorithm is independent of the actual implementation of *analyze-function*. We detail in Section 5.3 a direct implementation of *analyze-function* that efficiently computes the input partitions signature, but only for non-recursive functions. In Section 5.4, we propose an implementation of *analyze-function* based on a regular language invariant learning procedure that can analyze any function, including recursive functions. In our implementation (see Section 6), we combine the two versions to optimize the efficiency.

5.3 Non-recursive TRS

In the case of non-recursive TRSs, it is possible to directly find a solution to the type partition inference problem, without relying on any learning technique needed in the general case. We give the following implementation for the *analyze-function* algorithm that works in this specific settings.

This algorithm makes the distinction between two cases: constructor symbols for which no rules of $\mathcal{R}$ apply, and functional symbols for which at least one rule apply. The constructor case is handled using a *projection operator* whose idea is as follows: If a regular language $\mathcal{L}$ is shaped as $\{f(t_1, \dots, t_n) \mid t_1 \in \mathcal{L}_1, \dots, t_n \in \mathcal{L}_n\}$, then the projection of $\mathcal{L}$ on the symbol f at position i is $\mathcal{L}_i$. For instance, if we consider the language of lists of As and Bs defined by $\mathcal{L} = \{nil, cons(A, nil), cons(B, nil), cons(A, cons(A, nil)), \dots\}$, then $proj(\mathcal{L}, cons, 1)$ is $\{A, B\}$ and $proj(\mathcal{L}, cons, 2) = \mathcal{L}$. In our case, since a type basically represents a regular language, we directly define the projection on types. We then further extend this definition to partitions of types.

Definition 5.11 (Type projection). Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$. The projection of the type $\tau \in \Sigma^\#$ on a given symbol f at position i is

$$proj(\tau, f, i) = unify \ \{\tau_i \mid f(\tau_1, \dots, \tau_i, \dots, \tau_n) \rightarrow^* \tau \in \Delta^\#\}$$

where $unify$ is the type unification function. We write $unify(\{\tau_1, \dots, \tau_n\})$ for the most precise type τ w.r.t. $\preceq$ such that for all $i, 1 \leq i \leq n$, $\tau_i \preceq \tau$. [2]By convention, we note $proj(\tau, f, i) = \bot$ when the projection is not defined, that is when f is not recognized by τ in $\Delta^\#$. If we consider the type partition $T = \tau_1, \dots, \tau_n$, the projection of T on the symbol f at position i is defined as

$$proj(T, f, i) = remove_\bot\{proj(\tau_1, f, i), \dots, proj(\tau_n, f, i)\}$$

[2] Note that we can always build such unified type. In the worst case we can unify two arbitrary types by introducing a new abstract value $any^\#$ such that $\gamma(any^\#) = \mathcal{T}(\mathcal{F})$

```
function analyze-function
    input  : A TRS $\mathcal{R}$, an initial abstract domain $\Lambda_*$, a symbol $f \in \mathcal{F}^n$ and an output type
             partition $T$
    output : An abstraction $\mathcal{R}^\#$ and a partition signature $(T_1, \dots, T_n) \to T$ of $f$ in $\mathcal{R}^\#$
    Let $\mathcal{R}_f$ be the set of rules applicable on $f$;
    if $\mathcal{R}_f$ is empty then
        /* $f$ is a constructor symbol, for a value                */
        return $(proj(T, f, 1), \dots, proj(T, f, n))$
    else
        /* $f$ is a function symbol that can be rewritten          */
        Let $K \leftarrow |\mathcal{R}_f|$;
        foreach rule;
        $rule_k = f(p_1, \dots, p_n) \to r \in \mathcal{R}$ do
            Let $\Lambda_k, \mathcal{R}^\#_k, (T_{k,1}, \dots, T_{k,n}) \leftarrow$ rule-signature$(\mathcal{R}, \Lambda_*, rule_k, T)$;
        Let $\Lambda \leftarrow \Lambda_1 \cup \cdots \cup \Lambda_K$;
        Let $\mathcal{R}^\# \leftarrow \mathcal{R}^\#_1 \cup \cdots \cup \mathcal{R}^\#_K$;
        return merge-rules$(\Lambda, \mathcal{R}^\#, (T_{1,1}, \dots, T_{1,n}), \dots, (T_{K,1}, \dots, T_{K,n}))$
```

Algorithm 2: Non recursive symbol signature

where $remove_\perp$ filters out the $\perp$ element from the set. Note that $proj(T, f, i)$ is also a type partition.

The projection operator (Definition 5.11) is used in the algorithm to directly find the type partition signature of a constructor symbol. We can show that picking the projection of every sub-position of the constructor symbol on the target partition gives us the type partition signature of this symbol for this target type partition.

Lemma 5.12 (Projection on constructor). *Let $p = f(p_i, \dots, p_n)$ be a non functional pattern (without function symbol) and T a type partition. Then $(T_1, \dots, T_n) \to T$ where $T_i = proj(T_i, f, i)$ is a symbol signature of f.*

Proof. Let $(\tau_1, \dots, \tau_n)$ be a combination of $(T_1, \dots, T_n)$. Let $\tau \in T$ be the type such that for all i, $\tau_i = proj(\tau, f, i)$. Let π be a type environment such that $\Lambda^\#, \mathcal{R}^\#, \pi \vdash p_i : \tau_i$. By definition of the projection, we have $f(\tau_1, \dots, \tau_n) \to^*_{\Lambda^\#} \tau$. We can then conclude by construction of the type judgment rules that we have $\Lambda^\#, \mathcal{R}^\#, \pi \vdash p : \tau$. □

Finding the type partition signature of a functional symbol is a bit harder since it involves following the rewriting rules associated to this symbol. To do that that we first introduce the notion of "rule type partitions signature". Informally, for a given rule of a symbol, a rule type partitions signature is a type partitions signature for the symbol that is correct when considering this symbol's rule alone. This allows us to analyse every rule separately. We then show that we can merge all the rule signatures into one symbol signature.

Example 5.13. Consider the following rules encoding the *if-then-else*:

$$if(true, x, y) \to x \qquad\qquad if(false, x, y) \to y$$

Consider the type $nat^\#$ with two sub-types $even^\#$ and $odd^\#$ and the target type partition T defined as $T = \{even^\#, odd^\#\}$. The *rule signature* of the first rule for this target type partition would be $(\{true^\#, false^\#\}, \{even^\#, odd^\#\}, \{nat^\#\}) \to T$. For this rule, there is no constraints on y, so the most general partition $\{nat^\#\}$ is given. Conversely, the *rule signature* of the second rule would be $(\{\mathit{true}^\#, \mathit{false}^\#\}, \{\mathit{nat}^\#\}, \{\mathit{even}^\#, \mathit{odd}^\#\}) \to T$. We then merge both rule-signatures into one input partitions signature for *if* by using the product of each sub-partitions. The resulting input partitions signature is $(\{\mathit{true}^\#, \mathit{false}^\#\}, \{\mathit{even}^\#, \mathit{odd}^\#\}, \{\mathit{even}^\#, \mathit{odd}^\#\}) \to T$.

The formal definition of *rule signatures* and the demonstration on how to merge them into a single symbol signature are given in appendix of this paper. Note that this definition of *analyze-function* works for non-recursive functions. Despite the amount of proofs needed to prove it correct, this algorithm combined with *type-pattern* only involves some basic operations on tree automata to directly find a solution to the type partition inference problem. In the next section, we introduce a new definition of *analyze-function* intended for recursive functions.

5.4 Invariant learning

The main difficulty in functional program analysis is recursion, or in our case, the analysis of functional symbols defined with mutually recursive rewriting rules. In this section, we define an implementation of *analyze-function* based on an original invariant learning procedure. For a given (recursive function) symbol and target type partition, this procedure finds correct input regular languages partitions that completes the symbol's type partitions signature. It follows the standard outline of a counter-example guided abstraction refinement (CEGAR) procedure where a rough abstraction is iteratively refined using constraints learned from previous iterations. Those new constraints are defined by finding a *spurious* counter-example generated because of a faulty previous abstraction. Constraints are accumulated until no *spurious* counter-example can be found in which case the invariant has been found, or by finding a real counter-example. In this section we show how to use the *Tree Automata Completion Algorithm* [Genet and Rusu 2010] to adapt this family of techniques to Term Rewriting Systems and Regular Languages, allowing us to learn recursive symbol partitions signatures.

To find a type partition signature for a symbol f with the target type partition T, the procedure showed as Algorithm 3 computes a series of tree automata $\mathcal{A}_0^\#, \mathcal{A}_1^\#, \ldots$ according to the following outline: (1) We start by using the Tree Automata Completion Algorithm on $\mathcal{R}$ and an automaton $\mathcal{A}_0$ recognizing a finite subset $\mathcal{L}_0$ of the language $\mathcal{L} = \{f(t_1, \ldots, t_n) \mid t_1 \ldots t_n \in \mathit{GNF}(\mathcal{R})\}$. The result is an automaton $\mathcal{A}_0^*$ recognizing $\mathcal{R}^*(\mathcal{L}_0)$. This automaton is guaranteed to exist if $\mathcal{R}$ is terminating [Genet and Rusu 2010]. (2) We then check for any counter-example: any input term that violate the target type partition T by rewriting to two different types of the partition. For now, no abstraction has been done. If a counter-example is found in $\mathcal{A}_0^*$ recognizing $\mathcal{R}^*(\mathcal{L}_0)$, it is a real counter-example, no such signature exists for f and the property is disproved. Otherwise, using $\mathcal{A}_0^*$ and T, we build a set of disequality constraints over its states. (3) We then merge the states of $\mathcal{A}_0^*$ according to those disequality constraints to build an *abstraction* $\mathcal{A}_0^\#$ as the *smallest automaton* respecting those constraints. (4) If $\mathcal{A}_0^\#$ is complete and $\mathcal{R}$-closed (cf. Definition 5.15), then we know it contains a valid abstraction of $\mathcal{R}$ and Σ, and we can extract a signature for the symbol f. If not, we need to start over from (1) with a new automaton $\mathcal{A}_1$, recognizing $\mathcal{L}_1$ another finite subset of $\mathcal{L}$ such that $\mathcal{L}_1 \supset \mathcal{L}_0$. In Algorithm 3, this is done by the *grow* function which is defined Section 5.4.4. We continue until a counter-example is found, or a type partitions signature is found.

Example 5.14. Consider again the TRS defined by

$$\mathit{even}(0) \to \mathit{true} \qquad \mathit{odd}(0) \to \mathit{false}$$
$$\mathit{even}(s(x)) \to \mathit{odd}(x) \qquad \mathit{odd}(s(x)) \to \mathit{even}(x)$$

We want to find a partitions signature for the symbol *even* for the target type partition $T = \{\mathit{true}^\#, \mathit{false}^\#\}$. The language $\mathcal{L}$ generated by this symbol is $\mathcal{L} = \{\mathit{even}(n) \mid n \in \mathbb{N}\}$. We start (1) with $\mathcal{A}_0$ recognizing $\mathcal{L}_0 = \{\mathit{even}(0)\}$, and use the tree automata completion algorithm on it, which

```
function analyze-function
    input  : An input TRS $\mathcal{R}$, an initial abstraction $\Lambda_*$, a function symbol $f$, and a target
             type partition $T$.
    output : A type partitions signature $(T_1, \ldots, T_n) \to T$ for $f$.
    let $\mathcal{L} = \{f(t_1, \ldots, t_n) \mid t_1 \ldots t_n \in GNF(\mathcal{R})\}$;
    let $\mathcal{A}_0 = finite\text{-}subset(\mathcal{L})$;
    let $i = 0$;
    forever
        /* (1) Tree Automata Completion                              */
        let $\mathcal{A}_i^* = tree\text{-}automata\text{-}completion(\mathcal{A}_i, \mathcal{R})$;
        /* (2) Counter-example and constraints generation            */
        let $\phi = S(\mathcal{A}_i^*, T)$;
        if $\phi$ is unsatisfiable then return counter example ;
        /* (3) Abstraction                                           */
        let $\mathcal{A}_i^\# = \phi(\mathcal{A}_i^*)$;
        /* (4) Validity Check / Termination                          */
        if $\mathcal{A}_i^\#$ is $\mathcal{R}$-closed and GNF-complete then
            let $T_i = \{\tau_i \mid f(\ldots, \tau_i, \ldots) \to \tau \in \mathcal{R}^\#\}$;
            return $(T_1, \ldots, T_n) \to T$
        else
            $\mathcal{A}_{i+1} = grow(\mathcal{A}_i, \mathcal{L})$;
            $i = i + 1$;
```

Algorithm 3: Invariant learning procedure

gives us a new tree automaton $\mathcal{A}_0^*$ recognizing the reachable terms $\{even(0), true\}$ and defined by the following rules

$$(a) \qquad 0 \to q_0 \qquad even(q_0) \to q_{e0} \qquad true \to q_t \qquad q_t \to q_{e0}$$

There is (2) no violation of T yet. Let $S(\mathcal{A}_0^*, T)$ be the set of constraints to consider to (3) build an abstraction from $\mathcal{A}_0^*$, in which terms typed with different types of T are recognized by different states in the abstraction. For now, the only constraints to consider are well-typedness constraints: $S(\mathcal{A}_0^*, T) = \{q_0 \neq q_{e0}, q_t \neq q_0\}$. Indeed, q_0 recognizes a fragment of type $\mathbb{N}$ and q_t, q_{e0} a fragment of type *bool*, they must not be merged. We then use an SMT-solver to build the smallest renaming ϕ from Q to $\Sigma^\#$ respecting the given constraints. The result, $\phi(\mathcal{A}_0^*)$, is as follows (for the sake of readability we give comprehensible names to the new elements of $\Sigma^\#$):

$$0 \to nat^\# \qquad even(nat^\#) \to true^\# \qquad true \to true^\#$$

This automaton is not complete (4): the well-typed term $even(s(0))$ is not recognized. We hence start over (1) with $\mathcal{A}_1$ recognizing $\mathcal{L}_1 = \{even(0), even(s(0))\}$. After using the completion algorithm, $\mathcal{A}_1^*$ contains the transition set (a) plus the following new transitions:

$$(b) \quad s(q_0) \to q_1 \quad even(q_1) \to q_{e1} \quad odd(q_0) \to q_{e2} \quad false \to q_f \quad q_f \to q_{e2} \quad q_{e2} \to q_{e1}$$

Building the associated abstraction (3), we use the set of constraints $S(\mathcal{A}_1^*, T) = S(\mathcal{A}_0^*, T) \cup \{q_{e0} \neq q_{e1}, q_{e0} \neq q_{e2}, q_{e0} \neq q_f\} \cup \{q_t \neq q_{e1}, q_t \neq q_{e2}, q_t \neq q_f\}$ where we separate q_{e0}, q_t from q_{e1}, q_{e2}, q_f because they matches two different elements of the type partition T, respectively $true$ and $false$.

The resulting $\phi(\mathcal{A}_1^*)$ is

$$\begin{array}{rr} 0 \rightarrow nat^{\#} & s(nat^{\#}) \rightarrow nat^{\#} \\ true \rightarrow true^{\#} & false \rightarrow false^{\#} \\ even(nat^{\#}) \rightarrow true^{\#} & odd(nat^{\#}) \rightarrow false^{\#} \end{array}$$

This automaton is not $\mathcal{R}$-closed w.r.t. to the rule $odd(s(x)) \rightarrow even(x)$: if we instantiate x with $nat^{\#}$, since $even(nat^{\#})$ is recognized in $true^{\#}$ and $odd(s(nat^{\#}))$ in $false^{\#}$, for it to be $\mathcal{R}$-closed it should include the transition $true^{\#} \rightarrow false^{\#}$, which it does not. So we start again (1) with $\mathcal{A}_2$ recognizing $\mathcal{L}_2 = \{even(0), even(s(0)), odd(s(0))\}$ where $odd(s(0))$ has been generated from $odd(s(nat^{\#}))$. After using the completion algorithm, $\mathcal{A}_2^*$ contains transition sets (a), (b) and the new transition $odd(q_1) \rightarrow q_{e0}$. The associated set of constraints is $S(\mathcal{A}_2^*, T) = S(\mathcal{A}_1^*, T) \cup \{q_0 \neq q_1 \vee q_{e0} = q_{e1}\}$. The new constraint $q_0 \neq q_1$ is added because with the two transitions $odd(q_0) \rightarrow q_{e1}$ and $odd(q_1) \rightarrow q_{e0}$ if the abstraction chooses $q_0 = q_1$ then the resulting abstraction automaton would no longer be deterministic. The resulting $\phi(\mathcal{A}_2^*)$ is defined by

$$\begin{array}{rrr} 0 \rightarrow 0^{\#} & s(1^{\#}) \rightarrow 0^{\#} & s(0^{\#}) \rightarrow 1^{\#} \\ true \rightarrow true^{\#} & false \rightarrow false^{\#} & even(0^{\#}) \rightarrow true^{\#} \\ even(1^{\#}) \rightarrow false^{\#} & odd(0^{\#}) \rightarrow false^{\#} & odd(1^{\#}) \rightarrow true^{\#} \end{array}$$

This automaton is completed and $\mathcal{R}$-closed. It contains both an abstraction of $\mathcal{R}$ and of Σ. From it, we can extract a symbol signature for *even* with output partition $T = \{true, false\}$. The signature is $(\{0^{\#}, 1^{\#}\}) \rightarrow T$.

The remainder of this section details each step of one iteration of the procedure.

5.4.1 Tree Automata Completion. Let $\mathcal{R}$ be a functional TRS, and f the symbol of Σ we are looking a type partitions signature for. Each iteration i of the procedure starts by using the Tree Automata Completion algorithm to complete the automaton $\mathcal{A}_i$. This automaton is an ϵ-free tree automaton in which each state q recognizes exactly one term. The language recognized by $\mathcal{A}_i$ is a finite subset of where $\mathcal{L} = \{f(t_1, \dots, t_n) \mid t_1 \dots t_n \in \mathit{GNF}(\mathcal{R})\}$. A state q is final in $\mathcal{A}_i$ if it recognizes a term of $\mathcal{L}$. We write $\mathcal{A}_i^*$ for the output of this step of the procedure. It is a new tree automaton recognizing exactly $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_i))$ and having additional properties ensured by the Tree Automata Completion algorithm: $\mathcal{A}_i$ is syntactically included in $\mathcal{A}_i^*$, and for all state q, q' of $\mathcal{A}_i^*$ and term t (resp. t') recognized by q (resp. q'), if $t \rightarrow_{\mathcal{R}} t'$ then there exists a transition $q' \rightarrow q$ in $\mathcal{A}_i^*$ (t' is also recognized by q).

5.4.2 Counter-example finding and constraints generation. Since $\mathcal{A}_i^*$ converges to $\mathcal{R}^*(\mathcal{L})$, it can be used to search for a counter-example. Note that at this point, no abstraction has been made: a counter-example found in $\mathcal{A}_i^*$ is not spurious. Note also that the structure of a completed automaton allows us to easily build the rewriting path from an initial term of $\mathcal{L}$ to its faulty outcomes.

5.4.3 Abstraction generation. An abstraction $\mathcal{A}_i^{\#}$ is built from $\mathcal{A}_i^*$ by first computing a set $S(\mathcal{A}_i^*, T)$ of constraints over the states of $\mathcal{A}_i^*$ (which will depends on the target type partition T). For any tree automaton $\mathcal{A} = \langle \Sigma, Q, Q_f, \Delta \rangle$ and type partition T, $S(\mathcal{A}, T)$ is the smallest fixpoint such that:

$$q \neq q' \Leftarrow \exists \tau, \tau' \in T.\ q \in T_{\mathcal{A}}(\tau) \wedge q' \in T_{\mathcal{A}}(\tau') \wedge \tau \neq \tau' \tag{1}$$

$$q_1 \neq q_1' \vee \cdots \vee q_n \neq q_n \vee q = q' \Leftarrow f(q_1, \dots, q_n) \rightarrow q \in \mathcal{A} \wedge f(q_1', \dots, q_n') \rightarrow q' \in \mathcal{A} \tag{2}$$

$$v \neq v' \vee q = q' \Leftarrow v \rightarrow q \in A \wedge v' \rightarrow q' \in \mathcal{A} \tag{3}$$

where $T_{\mathcal{A}}(\tau) = \{ q_f \mid q_f \in Q_f \wedge \exists t \in \gamma(\tau).\ t \rightarrow^*_{\mathcal{A}} q_f \}$. In other words, $T_A(\tau)$ contains all the final states recognizing terms of type τ. The first type of constraints (1) ensures that the target type partition is respected: for any two final states q, q' (recognizing each a term of $\mathcal{L}$), if they rewrite into members of two different types of T, then we must have the constraint $q \neq q'$. The two other kind of constraints, (2) and (3), ensure determinism. We then use a SMT-solver to find *the smallest* renaming ϕ from Q to $\Sigma^\#$ such that $\mathcal{A}^\#_i = \phi(\mathcal{A}^*_i)$ satisfies $\phi(S(\mathcal{A}^*_i, T))$.

5.4.4 Termination. We stops the procedure when we detect that $\mathcal{A}^\#_i$ contains an abstraction of $\mathcal{R}$. It is the case when $\mathcal{A}^\#_i$ is $\mathcal{R}$-closed and *GNF*-complete w.r.t $\mathcal{R}$.

Definition 5.15 ($\mathcal{R}$-closed tree automaton). Let $\mathcal{R}$ be a term rewriting system and $\mathcal{A}$ a tree automaton. For all state q of $\mathcal{A}$, rule $l \rightarrow r$ of $\mathcal{R}$ and substitution σ of $\mathcal{X} \rightarrow Q$ such that $l\sigma \rightarrow^*_{\mathcal{A}} q$, the pair $\langle l\sigma, q\rangle$ is called a *critical pair*. It is *resolved* if there exists q' such that $r\sigma \rightarrow^*_{\mathcal{A}} q'$ and $q' = q$ or $q' \rightarrow q \in \mathcal{A}$. $\mathcal{A}$ is $\mathcal{R}$-closed if every critical pair is resolved.

Definition 5.16 (GNF-completeness). An automaton $\mathcal{A}$ is *GNF*-complete w.r.t $\mathcal{R}$ if for all symbols f used in the automaton, for every term $t = f(t_1, \ldots, t_n)$ where $t_i \in \mathit{GNF}(\mathcal{R})$ there exists a state q such that $t \rightarrow^*_{\mathcal{A}} q$. Note that in our case, $\mathit{GNF}(\mathcal{R})$ is easily computable since it is the language of values.

Before starting a new iteration of *analyze-function*, we build $\mathcal{A}_{i+1} = \mathit{grow}(\mathcal{A}_i, \mathcal{L})$. Here is a definition of this automaton:

(i) if $\mathcal{A}^\#_i$ contains an unresolved critical pair $\langle l\sigma, q\rangle$, then a rewriting path has not been taken into account yet. Let t be a term such that $t \rightarrow^*_{\mathcal{A}^\#_i} l\sigma$. Then, we add to $\mathcal{A}_{i+1}$ the necessary transitions to recognize t. This ensures that in the next iteration, this critical pair will be solved.

(ii) if $\mathcal{A}^\#_i$ has no unresolved critical pair but it is not *GNF*-complete, then we know that the set $E = \mathcal{L} \setminus \mathcal{L}(\mathcal{A}^\#_i)$ is not empty. Let t be one term of E having the smallest number of symbols. Then we add to $\mathcal{A}_{i+1}$ the necessary transitions to recognize t. This ensures that in the next iteration, $\mathcal{A}^\#_{i+1}$ recognizes it.

Finally, when *analyze-function* halts on an automaton $\mathcal{A}^\#_i = \langle \Sigma, Q, Q_f, \Delta\rangle$, it is *GNF*-complete, $\mathcal{R}$-closed, and $\mathcal{A}^\#_i$ contains an abstraction $\Lambda = \langle \Sigma^\#, \Delta^\#\rangle$ and an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ where $\Sigma^\#$ is Q the set of states of $\mathcal{A}^\#_i$, and where $\Delta^\#$ and $\mathcal{R}^\#$ are defined as

$$\mathcal{R}^\# = \{f(\tau_1, \ldots, \tau_n) \rightarrow \tau \mid f \in \mathcal{F} \wedge f(\tau_1, \ldots, \tau_n) \rightarrow \tau \in \Delta\} \tag{4}$$

$$\Delta^\# = \Delta \setminus \mathcal{R}^\# \tag{5}$$

Lemma 5.17 (The procedure outputs an abstraction of $\mathcal{R}$). *Let $\Sigma = C \cup \mathcal{F}$ be a ranked alphabet, with $\mathcal{R}$ a rewriting system, functional w.r.t. C and $\mathcal{F}$. Let $\mathcal{A}$ be a normalized, GNF-complete, and $\mathcal{R}$-closed automaton that is ϵ-deterministic (deterministic once ϵ-transitions are removed). Let ϕ be a renaming respecting the constraints $S(\mathcal{A}, T)$. Let $\mathcal{R}^\# = \{f(\tau_1, \ldots, \tau_n) \rightarrow \tau \mid f \in \mathcal{F} \wedge f(\tau_1, \ldots, \tau_n) \rightarrow \tau \in \Delta\}$. If the resulting automaton $\phi(\mathcal{A})$ is $\mathcal{R}^\#$-closed and is GNF-complete w.r.t $\mathcal{R}$, then $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$.*

Proof. First we prove that for all terms t and type τ such that $t \rightarrow^*_{\mathcal{A}} \tau$, for all k and all term u such that $t \rightarrow^k_{\mathcal{R}} u$ then $u \rightarrow^*_{\mathcal{A}} \tau$. We proceed by induction on k. First if $k = 0$. Then $t = u$ and since we already know that $t \rightarrow^k_{\mathcal{A}} \tau$, we have $u \rightarrow^*_{\mathcal{A}} \tau$. Second, if $k = k' + 1$. Then by definition there exists a rule $l \rightarrow r \in \mathcal{R}$, a substitution σ and a position p such that $t|_p = l\sigma$ and $t \rightarrow_{\mathcal{R}} t[r\sigma]_p \rightarrow^{k'}_{\mathcal{R}} u$. Since $t \rightarrow^*_{\mathcal{A}} \tau$ and $\mathcal{A}$ is normalized, there exists τ_p such that $t|_p \rightarrow^*_{\mathcal{A}} \tau_p$. Since

$\mathcal{A}$ is $\mathcal{R}$-closed and $t|_p \rightarrow_{\mathcal{R}} r\sigma$ this also means that we have $r\sigma \rightarrow^*_{\mathcal{A}} \tau_p$. Note that we hence have $t[r\sigma]_p \rightarrow^*_{\mathcal{A}} t[\tau_p]_p \rightarrow^*_{\mathcal{A}} \tau$. By induction hypothesis on k this means that $u \rightarrow^*_{\mathcal{A}} \tau$.

Now we prove that $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$. For soundness, let $f(\tau_1, \dots, \tau_n) \rightarrow \tau$ be a rule of $\mathcal{R}^\#$. Let $t = f(t_1, \dots, t_n)$ be such that for all i, $t_i \in \gamma(\tau_i)$. Let $u \in GNF(\mathcal{R})$ a term such that $t \rightarrow^*_{\mathcal{R}} u$. Then thanks to what we just proved, we know that $u \rightarrow^*_{\mathcal{A}} \tau$. Since u is irreducible, we deduce that $u \rightarrow^*_{\Delta^\#} \tau$. By definition this means that $u \in \gamma(\tau)$. To prove completeness, let $t = f(t_1, \dots, t_n)$ be such that for $i = 1 \dots n$, $t_i \in GNF(\mathcal{R})$. By assumption, we know that $\mathcal{A}$ is *GNF*-complete, thus there must exist a rule $f(\tau'_1, \dots, \tau'_n) \rightarrow \tau'$ such that $t_i \in \gamma(\tau'_i)$ for $i = 1 \dots n$. □

Theorem 5.18 (Correctness of the invariant learning procedure). *Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$, $\mathcal{R}^\#$ be the output of the invariant learning procedure of $\mathcal{R}$ with $\Lambda_* = \langle \Sigma^\#_*, \Delta^\#_* \rangle$, the pattern p and type partition T. Let Π be the set of all substitutions π such that $p\pi \rightarrow^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$. Then Λ, $\mathcal{R}$, $\widetilde{\Pi}$ is a solution to the type partition inference problem.*

Proof. First, since transitions are successively added to the automata $\mathcal{A}_i$, the facts $\Sigma^\# \supseteq \Sigma^\#_*$ and $\Delta^\# \supseteq \Delta^\#_*$ are ensured by construction of $\mathcal{A}_i$. Second, thanks to Lemma 5.17 we know that $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$. We need to show that this abstraction is complete w.r.t p and every $v^\#$ of T (cf. Definition 4.4). Consider $v^\# \in T$, a term $t \in \gamma(v^\#)$ and a (concrete) substitution σ such that $p\sigma \rightarrow^*_{\mathcal{R}} t$. Let $\mathcal{A}$ be the last automaton considered during the inference procedure. Since $\phi(\mathcal{A})$ is *GNF*-complete there must exist an (abstract) substitution π such that $\sigma \in \gamma(\pi)$. Since each term in $\mathcal{A}$ is recognized by a unique state, t will be abstracted in $\phi(\mathcal{A})$ by a unique abstract value. By definition of $S(\mathcal{A}, T)$ this abstract value is $v^\#$ [3]. Hence, if $p\pi \not\rightarrow^*_{\phi(\mathcal{A})} v^\#$ this means that $\phi(\mathcal{A})$ is not complete, which contradicts our hypothesis (otherwise the procedure would still continue). We have $p\pi \rightarrow^*_{\phi(\mathcal{A})} v^\#$ and by definition, $p\pi \not\rightarrow^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$. □

By Theorem 5.18, if we let T_i denote the set $\{\tau_i \mid f(\dots, \tau_i, \dots) \rightarrow \tau \in \mathcal{R}^\#\}$ then this procedure gives $(T_1, \dots, T_n) \rightarrow T$ as a correct signature of f in $\mathcal{R}^\#$ and $\Lambda^\#$. We can thus replace the *analyze-function* algorithm in *partition-inference* with this procedure when the input symbol is *recursive*, that is, its associated TRS fragment is recursive. It is worth noticing that this procedure provides two guarantees: (1) *regular completeness*, if there exists a regular abstraction $\mathcal{R}^\#$ providing a signature for f, then we will eventually find it; and (2) *completeness in refutation*, if there exists a counter-example, we will eventually find it.

Theorem 5.19 (Regular completeness). *If there exists a regular abstraction $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ of $\mathcal{T}(\Sigma)$ and a regular abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ providing a type partition signature for f, Algorithm 3 will eventually find it.*

Proof. In the following, given automata $\mathcal{A}$ and $\mathcal{A}'$, $\mathcal{A} \subseteq \mathcal{A}'$ denotes that transitions of $\mathcal{A}$ are included in transitions of $\mathcal{A}'$ modulo state renaming. In the same way, $\mathcal{A} = \mathcal{A}'$ means that transitions sets are equal modulo state renaming. If Λ and $\mathcal{R}^\#$ exists, then we can build $\mathcal{A}^\#$ defined with the union of $\Delta^\#$ and $\mathcal{R}^\#$. At each cycle i of the procedure, there exists a renaming ϕ respecting $S(\mathcal{A}^*_i, T)$ such that $\phi(\mathcal{A}^*_i) \subseteq \mathcal{A}^\#$. Since the procedure builds the *smallest* renaming ϕ' respecting $S(\mathcal{A}^*_i, T)$ (meaning that $\phi'(\mathcal{A}^*_i)$ has the smallest number of states), and since there is a finite number of automata for a given number of states, then we will eventually have $\phi = \phi'$. The procedure stops when $\phi'(\mathcal{A}^*_i)$ is *GNF*-complete, which means we cannot have $\phi'(\mathcal{A}^*_i) \subset \mathcal{A}^\#$, but only $\phi'(\mathcal{A}^*_i) = \mathcal{A}^\#$. □

[3] In practice, t could be abstracted into a more precise abstract value $w^\#$ such that $\gamma(w^\#) \subset \gamma(v^\#)$. In this rare case, an epsilon transition $w^\# \rightarrow v^\#$ is added to the automaton to retain the information. The rest of the proof is unchanged.

THEOREM 5.20 (COMPLETENESS IN REFUTATION). *For a given input function symbol f and target type partition T, if there exists a term $f(t_1, \ldots, t_n)$ that rewrites to two different elements of T, Algorithm 3 will eventually find it.*

PROOF. By adding new terms in the automaton $\mathcal{A}_i$ at each new cycle in a fair manner (smallest terms first in step (ii) of *grow* function), we guarantee that at some point we will add the counter-example. It will be detected as a counter-example after the tree automata completion phase. □

Recall that our overall goal is to prove safety properties on programs, i.e., properties of the form $t \not\to^* \mathit{false}$. For this, we can identify a subset of *forbidden types* (such as *false*$^\#$) in the target type partition, and define a counter-example as a term t typable with one of those forbidden types. In this way we can turn our type inference algorithm into a verification tool for safety properties, as illustrated in the next section.

6 EXPERIMENTS

This section details our implementation [Haudebourg and Genet 2020] of the verification technique developed in this paper and the associated experimental results [Experiments 2020]. It consists of an OCaml program along with several libraries able to resolve together the regular language type inference problem from an input term rewriting system $\mathcal{R}$, pattern p and target type partition T. It outputs an abstraction Λ and $\mathcal{R}^\#$ and all the possible type substitutions π with the associated $\tau \in T$ such that $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$. This allows us to verify complex properties on programs by expressing the property using a predicate defined in the program itself, and verifying that it can never be typed with *false*$^\#$. For instance to prove that a list sorting function *sort* is correct (in the sense that the output list is sorted), we first define a *sorted* predicate as

$$\begin{aligned} sorted(nil) &\to true \\ sorted(cons(x, nil)) &\to true \\ sorted(cons(x, cons(y, z))) &\to if(leq(x, y), sorted(cons(y, z)), false) \end{aligned}$$

and use our implementation with the input term $sorted(sort(x))$ with the target type partition $\{true^\#, false^\#\}$. If all the output type substitutions π are such that $\pi \vdash sorted(sort(x)) : true^\#$, then the property is verified.

6.1 Implementation details

The implementation [Haudebourg and Genet 2020] follows the theoretical algorithms presented in this paper with some optimisations and limitations. We describe here the various specificities of the implementation.

Preliminary typing phases. To simplify our argumentation in this paper, we required every type partition to be a partition of $GNF(\mathcal{R})$, the entire set of possible values. In practice, trying to compute type partitions over $GNF(\mathcal{R})$ would be both inefficient and unnecessary since in most modern programming languages functions can only take given subsets of values (types) as parameters. For this reason, our implementation is equipped with a preliminary Hindley-Milner type inference phase [Hindley 1969; Milner 1978] with let-polymorphism. The first-order types used by this inference phase are defined by the user as an input tree automaton where each state is a type. From the input pattern, the term rewriting system is then *monomorphized* so that we know the input domain of every function call, and thus the domain of each type partition we are looking for. This greatly improves the performances of the analysis.

Constants subtyping phase. Consider the simple term rewriting system defining the equality predicate on natural numbers

$$eq(0,0) \to true \qquad eq(s(x),0) \to \mathit{false} \qquad eq(0,s(y)) \to \mathit{false} \qquad eq(s(x),s(y)) \to eq(x,y)$$

together with the initial abstract domain $\{true^\#, \mathit{false}^\#, nat^\#\}$. The problem of finding the signature of *eq* for the target partition $\{true^\#, \mathit{false}^\#\}$ is not regular if we consider the input domains of the arguments of *eq* to be $nat^\#$. In practice however, if *eq* is applied on a constant value, such as in $eq(x,s(0))$, we can reduce the domain of the second parameter by adding an abstract value $1^\#$ abstracting $s(0)$ in the initial abstract domain. The only possible partition of $1^\#$ is $\{1^\#\}$, which transform the problem of finding the (type partition) signature of *eq* for $\{true^\#, \mathit{false}^\#\}$ into a regular problem. Its solution is $(\{2+^\#, 1^\#, 0^\#\}, \{1^\#\})$. In our implementation, in addition to the first-order types given by the user, we build precise types for the constant values used in the program. This allows us to find a solution to otherwise irregular problems. It is however a prototypical optimisation that must be used with caution since it may lead to a computational overhead.

Counter-examples. When using our implementation to type a given input pattern with a given type partition, it is possible to specify that some types in the partition are *invalid*. For instance, while typing $sorted(sort(L))$ with the partition $\{true^\#, \mathit{false}^\#\}$, if the *sort* function is correct we expect the pattern to not be typable by $\mathit{false}^\#$. In this case, once the analysis is finished, if we find a way to type the pattern with $\mathit{false}^\#$, then we are able to generate a counter example: an instantiation of the input pattern that rewrites to *false*. Note that this implementation choice has two drawbacks: (1) Since the "forbidden type" information is only used once the analysis is done, if there is no regular types that satisfies the target type partition, our implementation may diverge even though there is a counter example to the desired property. (2) In the best case, even if everything is regular, this may still delay the finding of counter examples. It is possible to insert the notion of *forbidden type* into the analysis to avoid these issues, although this would require some adaptations of the algorithms presented in this paper.

6.2 Test suite

We tested our implementation over a collection of more than 80 problems [Haudebourg and Genet 2020] coming from Timbuk 3 [Genet et al. 2001], some regular problems from the MoCHi test suite [Kobayashi et al. 2011a] and some original challenges created for the occasion inspired by Tons of Inductive Problems [Claessen et al. 2015]. We expect some problem instances to be similar to the ones used in[Matsumoto et al. 2015], however this cannot be verified since the test suite used in this paper is not publicly available. The test suite is composed of a variety of problems over first-order and higher-order tree-processing functional programs including:

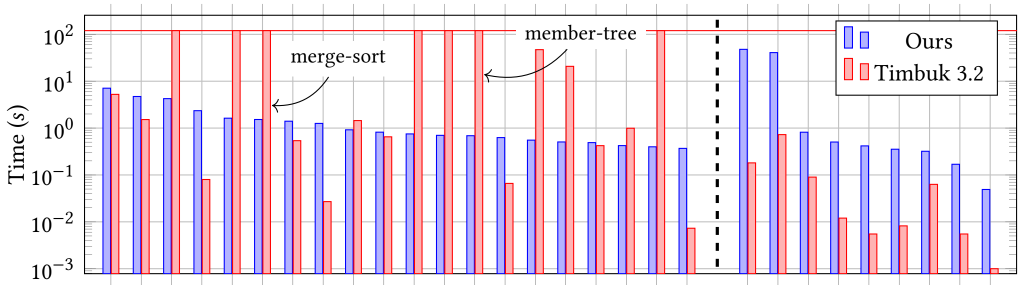
- Positive tests: regular properties intended to be proved by our implementation. This includes properties such as $\forall L. sorted(merge\text{-}sort(L)) = true$ where L is a list of As and Bs, or $\forall X,T.\ member(X,T) \iff member(X, mirror(T)))$ where T is a binary tree, etc.
- Negative tests: false properties or buggy programs, where there exists a counter example to the target property.
- Typing challenges where there is no property to verify but only regular language types to find. For instance, using the input pattern $only\text{-}As(L)$ and the target type partition $\{true^\#, false^\#\}$, our implementation should type L with either the regular language type "lists of As" or "lists with Bs".
- Intentionally non-regular problems, for which we expect the invariant learning procedure to diverge. For instance, $\forall N : nat.\ N = N$ is not a regular property.

Note that even if some of the problems in our test suite are taken from the MoCHi test suite [Kobayashi et al. 2011a], MoCHi does not handle regular languages in general and does not target the same family of properties.

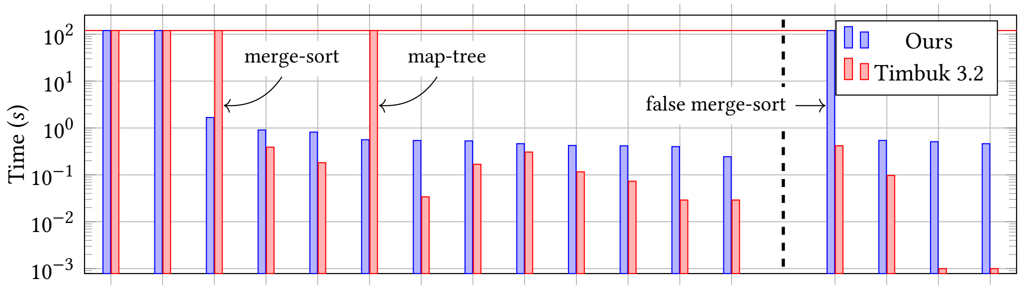
6.3 Experimental results

We compared the performances of our implementation over our test suite against Timbuk 3 which is, to our knowledge, the only higher-order tree-processing program verification tool that is publicly available. In particular, we have not been able to compare to the regular-complete verification procedure presented in [Matsumoto et al. 2015] which does not offer a public implementation, even if it also targets regular properties. The following graphs show the time performances (averaged over 10 executions) of the two implementations over the compatible regular test instances on a Intel® i7-7600U CPU, 4 2.80GHz cores. Positive instances (on the left) and negative instances (on the right) are separated by a dashed line. A timeout is set at 120 seconds (red line).

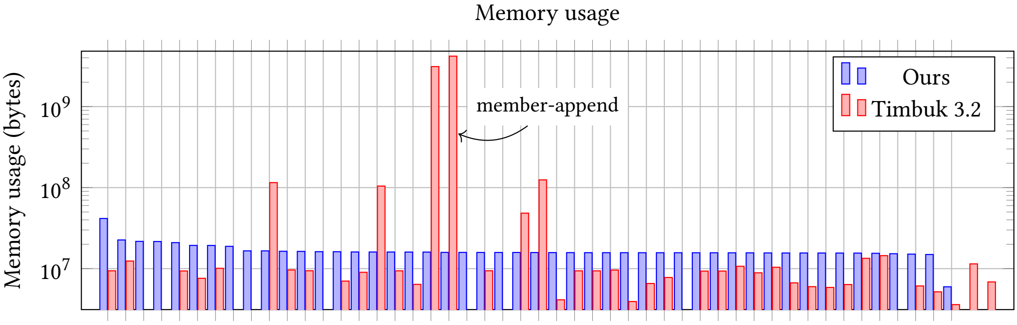
First-order programs



Higher-order programs



This shows that when it succeeds, Timbuk 3 is in average faster than our implementation. This is expected since our preliminary typing phase and subtyping phase has a cost. However this also shows that in many cases, even on first-order programs our implementation terminates where Timbuk 3 diverges. In particular this is the case for the non trivial *merge-sort* algorithm for which we successfully identify the regular language of sorted lists. This is also the case for binary-tree processing algorithms such as *member-tree* where we automatically show that it is invariant by mirroring. We see the same tendency on higher-order programs where Timbuk 3 also diverges on the merge-sort algorithm (where this time the comparison operator is a parameter of the sorting function), and on binary-tree processing programs such as *map-tree*, where we succeed. Interestingly, our implementation is unable to find a counter-example to a *false merge-sort* property, where we try to check that $\forall L.\ sorted\ (\leq)\ (merge\text{-}sort\ (\geq)\ L)$, which is wrong. Timbuk 3 is able to find a counter example, where we reach a timeout. As discussed in Section 6.1 this is due to the fact that, in our current modular implementation, counter-examples are searched once the whole analysis is finished. Another feature of our implementation is its consistent memory usage:



This graph shows that, thanks to modularization, the memory footprint our our implementation stays low even on time consuming instances. This is an improvement compared to the non-modular Timbuk 3, where on some problems the memory usage can grow up to several GB of data. The difference is especially visible for instance on the *member-append* problem where the goal is to verify that for all lists L_1, L_2 and element X, $(\mathit{member}\ X\ (\mathit{append}\ L_1\ L_2)) \iff ((\mathit{member}\ X\ L_1) \vee (\mathit{member}\ X\ L_2))$.

7 RELATED WORK

Most of the preexisting techniques for higher-order function verification techniques currently fall in two categories: one is expressive enough to verify a wide range of properties at the cost of requiring annotations on the program, the other require less or no annotations but does not play well with algebraic data types.

Type-system based techniques. In many ways this work has been inspired by Liquid Types [Rondon et al. 2008; Vazou 2016; Vazou et al. 2013, 2014], Bounded Refinement Types [Vazou et al. 2015], and Set-Theoretic Types [Castagna et al. 2014, 2016], as a way to enrich the type-system of the language to verify non-trivial properties on higher-order programs. These techniques however fall in the first category. The user still has to annotate the program with sometimes complex type annotations, sometimes equivalent to straightforward intermediate lemmas to help the type checker. In our case, at the price of a less expressive power, the only needed information is an initial term associated to a target type partition.

Algebraic data types. The most advanced fully automated techniques such as Kobayashi's [Champion et al. 2018; Kobayashi et al. 2011b; Sato and Kobayashi 2017], implemented in MoCHi [Kobayashi et al. 2011a], focus on relational properties over integers in the detriment of algebraic data types. Some efforts has been made to handle algebraic data types such as lists by encoding them using a function mapping each index to its value. However, this makes the program specification more intricate and degrades the performances of the analysis. On the opposite, our technique focuses on algebraic data types and cannot handle relational properties, which is complementary to MoCHi.

Invariant learning. The interest around the fully automated verification of tree processing programs can be traced back to Jones [Jones and Andersen 2007], followed by a long lineage of model-checking techniques with Ong's Higher-Order Recursion Schemes [Kobayashi 2009; Ong 2006] and Pattern Matching Recursion Scheme [Ong and Ramsay 2011] with, in parallel, Genet's Tree Automata Completion based technique [Genet 2016; Genet and Rusu 2010]. Similarly to our work in this paper, the overall goal of these technique has been to find ways of building finite abstractions of the behavior of a program able to prove a given property using various theoretical constructions. Since the beginning, the key difficulty was the inference of the recursive functions invariants. Until [Matsumoto et al. 2015], no completeness guaranties were provided. In most cases, even simple recursive functions can make these techniques diverge in a hopeless effort of successive abstraction refinements. Contrary to these previous techniques, and similarly to [Matsumoto et al. 2015], our invariant learning procedure ensure *regular completeness* and *completeness in refutation* (cf. Section 5.19). The core of the invariant learning procedure presented in this paper is greatly inspired by various Counter Example Guided Abstraction Refinement (CEGAR) techniques [Champion et al. 2018; Kobayashi et al. 2011b; Matsumoto et al. 2015; Ong and Ramsay 2011], however it is worth noticing some advantages of this version: usually, refining the abstraction is done by finding a *spurious* counter-example in the currently considered abstraction which can be computationally expensive for various reasons. In our case, refinement is done from the exact computation of a fragment of $\mathcal{R}^*(\mathcal{L})$. Executions paths are given "for free" in the computed fragment, at the price of requiring $\mathcal{R}$ to be terminating. This also gives us real counter-examples as easily.

Term Rewriting Systems and Tree Automata. Term Rewriting Systems and Tree Automata gives a natural framework to model higher-order functional programs and algebraic data types, allowing us to analyse complex programs without the need of transforming it beforehand into a completely different representation. This is to contrast with related works such as [Champion et al. 2018; Kobayashi et al. 2010] where a side effect of the transformation is that it makes it difficult to relate the verification result to the original program. Besides, the development of the Tree Automata Completion algorithm [Genet 2016; Genet and Rusu 2010] over the years makes this framework an appealing candidate for fully automatic program verification of higher-order programs. It has already been successfully used in program verification with Timbuk [Genet et al. 2018], but just like its model-checking counterparts at the time, with the lack of a complete invariant learning procedure. Our approach adapts this technique with actual invariant learning capabilities. Since our approach rely on the same framework, expressivity of the two approaches are close. In particular, our technique covers most of Timbuk's benchmarks but is also able to prove properties on which Timbuk fails, like the merge sort algorithm.

8 CONCLUSION AND FUTURE WORK

Contribution. We have developed a regular language type inference procedure on top of a regular abstract interpretation of term rewriting systems. This allows us to automatically verify safety properties on higher-order tree-processing functional programs. This improves on existing verification techniques based on type annotations which generally require a lot of expertise to determine all the necessary annotations to carry out the proof.

The type inference mechanism uses type partitions to reduce the complexity of the underlying algorithms. For a given input pattern and target type partition, we are able to associate type partitions to the variables of the pattern that describe how the pattern instances rewrites to elements of the target partitions. Using a type system allows for modularity: a type partition signature is attached to each symbol, which summarize the behavior of associated rewriting rules. The type partition signature can be inferred independently for independent functions. It can be inferred directly for non recursive functions. Recursive functions are handled using a novel invariant learning procedure based on the tree automata completion algorithm, following the precepts of a counter-example guided abstraction refinement procedure (CEGAR). This particular variant of CEGAR allows us to refine the abstractions without the need of a complete spurious counter-example rewriting sequence, at the price of requiring the input TRS to be terminating. The resulting procedure is regularly-complete and complete in refutation, meaning that if it is possible to give a regular language type to a term then we will eventually find it, and if there is no possible type (regular or not) then we will eventually find a counter-example. Our implementation of this technique shows encouraging performances. It is able to verify properties that were not covered by previous similar techniques. In the following we list some possible further improvements.

Inferring invariants for refinement type techniques. Using a more restricted type language helps us to enjoy more automation for type inference. In particular, annotations for intermediate functions are automatically inferred. For instance, from a TRS encoding the insertion sort algorithm, and the final term $sorted(sort(x))$, we are able to automatically infer the type annotation for the insert function as $insert(any^{\#}, sorted^{\#}) \rightarrow sorted^{\#} \in \mathcal{R}^{\#}$, where the abstract value $sorted^{\#}$ is inferred from the side analysis of the $sorted$ symbol: $sorted(sorted^{\#}) \rightarrow true^{\#} \in \mathcal{R}^{\#}$. This annotation reveals a necessary intermediate proof step: we first need to prove that when inserting *any* element into a *sorted* list, we obtain a *sorted* list. We believe that starting from the two above generated rules of $\mathcal{R}^{\#}$ it is possible to infer the adequate type annotations for F* or Liquid Types and automate the proofs, i.e., infer the annotation $insert(X : int, Y : \{l : int\ list \mid sorted(l)\}) : \{l : int\ list \mid sorted(l)\}$.

Extension to properties on lazy functional programs. Jones' abstraction technique [Jones and Andersen 2007] can prove properties on functional programs using call-by-name evaluation. With our current invariant inference procedure we need for the input program to be terminating. However, this inference part is not limited to call-by-value evaluation, as it is in [Jones and Andersen 2007]. We could broaden the image computation technique so that *unfinished* non terminating computations are taken into account. We experimented with this and there are some interesting cases, like call-by-value evaluations, for which this is sufficient to build a correct abstraction automaton.

Extension to domains mixing numerical values and data structures. There are known abstraction inference techniques for domains mixing structures and base types [Le Gall and Jeannet 2007], e.g., queues of integers. These techniques rely on an extension of word automata called lattice automata. There exists a similar extension for tree automata called lattice tree automata making it possible to abstract recursive structures containing, e.g., integer values [Genet et al. 2013].

Extension with regular relations. The main limitation of this technique is the impossibility to express or verify relational properties. There exists ways to use tree automata to represent relations using, for instance, automatic structures [Blumensath and Grädel 2000; Khoussainov and Nerode 1995]. The idea is to use one automaton to represent multiple terms at once, or *convolution of terms*. For instance, the equality relation can be recognized by the two automaton transitions $0 \star 0 \rightarrow q$ and $s \star s(q) \rightarrow q$, where $\star$ is a *convolution operator*. Checking if two terms t_1, t_2 are equals is then equivalent to testing if the convoluted term $t_1 \star t_2$ is recognized by the automaton above. Convolutions introduces however new challenges that remains to be solved in order to integrate them into this technique.

Acknowledgements

The authors would like to thank the anonymous referees for their detailed suggestions.

A MERGING SUBTERMS SOLUTIONS

This refers to the *merge* function used in *type-pattern* algorithm (cf. Algorithm 1, Section 4.2).

Lemma A.1 (Merge). *Let $p = f(p_1, \ldots, p_n)$. Let $(T_1, \ldots, T_n)$ be a signature of the symbol f for T in $\Lambda_* = \langle \Sigma^{\#}_*, \Delta^{\#}_* \rangle$, $R^{\#}_*$. Assume we have $\Lambda_i, R^{\#}_i, \widetilde{\Pi_i}$ a solution to the type partition inference problem with $\mathcal{R}, \Lambda_*, p_i, T_i$. If $\Lambda = \langle \Sigma^{\#}, \Delta^{\#} \rangle, \mathcal{R}^{\#}$ and $\widetilde{\Pi}$ (the output of merge) are the smallest sets such that (1) $\Sigma^{\#} \supseteq \Sigma^{\#}_1 \cup \cdots \cup \Sigma^{\#}_n$; (2) $\Delta^{\#} \supseteq \Delta^{\#}_1 \cup \cdots \cup \Delta^{\#}_n$; (3) $\mathcal{R}^{\#} = \mathcal{R}^{\#}_* \cup \mathcal{R}^{\#}_1 \cup \ldots \cup \mathcal{R}^{\#}_n$; (4) $\widetilde{\Pi} = \widetilde{\Pi}_1 \cup \ldots \cup \widetilde{\Pi}_n$*

is defined in Λ. *Then* $\Lambda, \mathcal{R}^\#$ *and* $\widetilde{\Pi}$ *are a solution to the type partition inference problem of* p *with* T *in* $\Lambda_*, \mathcal{R}$.

Proof. We prove that for all $\pi \in \Pi$, there exists a type $\tau \in T$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash p : \tau$. Let $\pi \in \Pi$. By definition of the union of type partition environnements, for each i, for each $\pi_i \in \Pi_i$, for each variable $x \in Var(p_i)$ we have either $\pi(x) = \pi_i(x)$ or there exists an epsilon transition $\pi(x) \rightarrow \pi_i(x) \in \Delta^\#$. Hence we have $\tau_i \in T_i$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash p_i : \tau_i$. Now by definition, since $(T_1, \ldots, T_n)$ is a signature of the symbol f for T in $\mathcal{R}^\#_*$ with $\mathcal{R}^\#_* \subseteq \mathcal{R}^\#$, we have that there exists $\tau \in T$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash p : \tau$. □

B CORRECTNESS OF THE *TYPE-PATTERN* ALGORITHM

This refers to the *type-pattern* algorithm introduced Section 4.2.

Theorem B.1. *Let* $\mathcal{R}$ *be a TRS,* $\Lambda_* = \langle \Sigma^\#_*, \Delta^\#_* \rangle$ *be an initial abstract domain,* p *a pattern with* T *a partition of* $\Sigma^\#_*$. *If* $\Lambda, \mathcal{R}^\#, \widetilde{\Pi} =$ *type-pattern*$(\mathcal{R}, \Lambda, p, T)$ *then it is a solution to the type partition inference problem of* p *with* T, Λ_* *and* $\mathcal{R}$.

Proof. By induction on the pattern p. If $p = x$, then the output of *type-pattern* is $\Lambda = \Lambda_*$, $\mathcal{R}^\# = \emptyset$ and $\widetilde{\Pi} = \{x \mapsto T\}$. We have (1) $\Sigma^\# \subseteq \Sigma^\#_*$ by construction (2) $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$ since it is empty (3) for all $\pi \in \Pi$, if we take $\tau = \pi(x)$ then $\pi, \Lambda, \mathcal{R}^\# \vdash x : \pi$. In the other case, if $p = f(p_1, \ldots, p_n)$, then we have $\Lambda', \mathcal{R}'^\#, (T_1, \ldots T_n) =$ analyze-function$(\mathcal{R}, \Lambda_*, f, T)$. Since we assume that *analyze-function* is correct, we known that $(T_1, \ldots, T_n)$ is the symbol signature of f in $\Lambda', \mathcal{R}'^\#$. We write $\Sigma^\#_i, \Lambda^\#_i, R^\#_i$ and $\widetilde{\Pi}_i$ for the outputs of *type-pattern*$(R, \Lambda', R'^\#, p_i, T_i)$ for all i. By hypothesis of induction, we know those are solutions to the type partition inference problem with $R, \Lambda', R'^\#, p_i, T_i$ as inputs. Then by definition of *merge* and Lemma A.1, we conclude that the returned $\Sigma^\#, \Lambda, R^\#, Pi^{-1}$ are solutions to the type partition inference problem of p with Λ_*, R and T. □

C COMPUTATION OF RULE SIGNATURES

The following gives the formal definition of *rule signatures* and shows how to merge those rules signatures into a symbol signature for a given target type partition. Note that we do not explicitly give an implementation of the *merge-rules* function used in the algorithm and assume it strictly follows the merge instructions given by Lemma C.2.

Definition C.1 (Rule type partitions signature). Let R be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstract domain, and $R^\#$ an abstraction of R defined over them. Let T be a type partition over $\Sigma^\#$. Let $l \rightarrow r$ be a rule of R such that $l = f(p_1, \ldots, p_n)$. We say that $(T_1, \ldots, T_n) \rightarrow T$ is the typr partitions signature for this rule in $\Lambda, R^\#$ if for all $\tau_1, \ldots, \tau_n$ where $\tau_i \in T_i$ there exists $\tau \in T$ such that for all π such that for all i, $\pi, \Sigma^\#, \Lambda^\#, R^\# \vdash p_i : \tau_i$ then we have $\pi, \Sigma^\#, \Lambda^\#, R^\# \vdash r : \tau$.

Lemma C.2 (From rule signatures to symbol signatures). *Let* $\mathcal{R}$ *be a TRS,* $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ *an abstract domain, and* $\mathcal{R}^\#$ *an abstraction of* R *defined over them. Let* T *be a type partition over* $\Sigma^\#$ *and* f *a symbol associated with a non-empty set of rules* $R_f \subseteq R$. *Let's index each rule of* R_f *by* k *from* 1 *to* $|R_f|$ *so that each rule is of the form* $f(p_{k,1}, \ldots, p_{k,n}) \rightarrow r_k$. *For all rule* k *let's note* $(T_{k,1}, \ldots, T_{k,n})$ *the input signature of this rule for* T *in* $\Lambda, R^\#$. *Let's define* $(T_1, \ldots, T_n)$ *such that*

$$\forall i, 1 \leq i \leq n. \quad T_i = T_{1,i} \otimes \ldots \otimes T_{|R_f|,i}$$

Let's note $T_{k,i}(\tau)$ *the type* $\tau' \in T_{k,i}$ *such that* $\tau \leq \tau'$. *For all* $\tau \in T_i$, *by definition this type should exists and be unique. Let's also note* $\Pi_k(\tau_1, \ldots, \tau_n) = \{ \pi \mid \forall i. \quad \pi, \Sigma^\#, \Lambda^\#, R^\# \vdash p_{k,i} : T_{k,i}(\tau_i) \}$. *If we define* $\mathcal{R}'^\#$ *such that*

$$\mathcal{R}'^\# = \mathcal{R}^\# \cup \{ f(\tau_1, \ldots, \tau_n) \rightarrow \tau \mid \forall i. \tau_i \in T_i \quad \wedge \quad \forall k. \forall \pi \in \Pi_k(\tau_1, \ldots, \tau_n). \ \pi, \Lambda, R^\# \vdash r_k : \tau \}$$

then $\mathcal{R}'^\#$ is an abstraction of $\mathcal{R}$ such that $(T_1, \dots, T_n) \to T$ is the type partitions signature of the symbol f.

Proof. For all $f'(\tau_1, \dots, \tau_n) \to \tau \in \mathcal{R}'^\#$. If $f \neq f'$, soundness and completeness are ensured by including $\mathcal{R}^\#$ in $\mathcal{R}'^\#$. Otherwise if $f = f'$, we first show that $\mathcal{R}'^\#$ is a sound abstraction of $\mathcal{R}$. Let $t = f(t_1, \dots, t_n)$ such that $t_i \in \gamma(\tau_i)$. Let's consider $u \in IRR(R)$ such that $t \to^*_{\mathcal{R}} u$. Since we consider functional TRSs, there exists a rule $f(p_{k,1}, \dots, p_{k,n}) \to r_k \in \mathcal{R}$ and a substitution $\sigma : X \mapsto IRR(\mathcal{R})$ such that $t_i = p_{k,i}\sigma$ and $t \to_{\mathcal{R}} r\sigma \to^*_{\mathcal{R}} u$. By definition we have $p_{k,i}\sigma \in \gamma(\tau)$, but also by construction, $p_{k,i}\sigma \in \gamma(T_{k,i}(\tau_i))$. Hence by definition there exists π such that for all i, $\pi, \Lambda, R^\# \vdash p_{k,i} : T_{k,i}(\tau_i)$. Note that by construction, $\pi \in \Pi_k(\tau_1, \dots, \tau_n)$. Since $(T_{k,1}(\tau_1), \dots, T_{k,n}(\tau_n))$ is a combination of the rule signature $(T_{k,1}, \dots, T_{k,n}) \to T$ of the rule k with $\Lambda, R^\#$, we know there exist a type $\tau \in T$ such that $\pi, \Lambda, R^\# \vdash r_k : \tau$. Since $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$ for Λ, by definition this means that $u \in \gamma(\tau)$. Now let's show $\mathcal{R}'^\#$ is a complete abstraction of $\mathcal{R}$. Let's consider any $(\tau_1, \dots, \tau_n)$ where $\tau_i \in T_i$. First note that $\mathcal{R}_f$ is not empty. Since the TRS is complete, there exists at least one rule f such that $\Pi_k(\tau_1, \dots, \tau_n)$ is not empty. Since $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$, there exists a unique τ such that for all rule k and all $\pi \in \Pi_k(\tau_1, \dots, \tau_n)$, $\pi, \Lambda, R^\# \vdash r_k : \tau$. Then by definition there exists the rule $f(\tau_1, \dots, \tau_n) \to \tau$ in $R'^\#$. □

The last missing piece left is the *rule-signature* algorithm that will allow us to actually compute the signature of each rule. For the considered rule $l \to r$, it will type the right-hand-side r of the rule in order to find the type partition of each variable (for this rule), and deduce a type for the left-hand-side from the found type partitions.

```
function rule-signature
    input  : A TRS $\mathcal{R}$, an initial abstract domain $\Lambda_*$, a rule $f(p_1, \dots, p_n) \to r \in \mathcal{R}$ and an
             output type partition $T$
    output : An abstraction $\mathcal{R}^\#$ and the rule signature $(T_1, \dots, T_n) \to T$ of the given rule
             in $\mathcal{R}^\#$
    Let $\Lambda, \mathcal{R}^\#, \widetilde{\Pi} \leftarrow$ type-pattern($\mathcal{R}, \Lambda_*, r, T$);
    foreach sub-pattern $p_i$ do
        Let $\Lambda_i, T_i \leftarrow$ bottom-up-type-pattern($\Lambda, p_i, \widetilde{\Pi}$);
    Let $\Lambda' \leftarrow \Lambda_1 \cup \dots \cup \Lambda_n$;
    return $\Lambda', \mathcal{R}^\#, (T_1, \dots, T_n)$
```

The goal of the *bottom-up-type-pattern* function is to give a fitting type partition to the left-hand-side of the rule from a given type partition environment. This type can easily be built by typing the pattern from bottom to top.

Example C.3 (Bottom-up typing). Let's imagine we wish to type the irreducible pattern $cons(s(x), y)$ with the type partition environment $\widetilde{\Pi} = \{x \mapsto \{0^\#, N{+}^\#\}, y \mapsto \{[nat\ list]^\#\}\}$ where $\Delta^\#$ is

$$\begin{array}{rr} 0 \to 0^\# & nil \to [nat\ list]^\# \\ s(0^\#) \to N{+}^\# & cons(0^\#, [nat\ list]^\#) \to [nat\ list]^\# \\ s(N{+}^\#) \to N{+}^\# & cons(N{+}^\#, [nat\ list]^\#) \to [nat\ list]^\# \end{array}$$

Since $\widetilde{\Pi}(x) = \{0^\#, N{+}^\#\}$ we can create two new types $[s(0)]^\#$ and $[s(N{+})]^\#$ in $\Sigma^\#$ with the associated rules $s(0^\#) \to [1]^\#$ and $s(N{+}^\#) \to [2{+}]^\#$ in $\Delta^\#$ such that a valid type partition for $s(x)$ is $\{[1]^\#, [2{+}]^\#\}$. We also need to add the sub-typing rules $[1]^\# \to N{+}^\#$ and $[2{+}]^\# \to N{+}^\#$ in $\Delta^\#$. Now that we have a partition for $s(x)$, we can continue up and type the whole pattern. This gives us the partition $\{[cons(1, nat\ list)]^\#, [cons(2+, nat\ list)]^\#\}$ associated to the rules

$$cons([1]^\#, [nat\ list]^\#) \longrightarrow [cons(1, nat\ list)]^\#$$
$$cons([2+]^\#, [nat\ list]^\#) \longrightarrow [cons(2+, nat\ list)]^\#$$

The actual specification of the *bottom-up-type-pattern* function is given by the following definition.

Definition C.4. Bottom-up typing Let $\mathcal{R}$ be a functional TRS. Let Λ be an abstract domain and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ defined over it. Let p be an *irreducible* pattern and $\widetilde{\Pi}$ a type partition environment. Let's name $\widetilde{\Pi}|_p = \{x \mapsto \widetilde{\Pi}(x) \mid x \in Var(p)\}$ the projection of $\widetilde{\Pi}$ on the variables of p (we discard other variables). The function *bottom-up-type-pattern* returns Λ', T such that there exists a bijection $\xi : \Pi|_p \mapsto T$ such that for all $\pi \in \Pi|_p, \quad \pi, \Lambda, \mathcal{R}^\# \vdash p : \tau(\pi)$.

From this definition, it is possible to show that the *rule-signature* algorithm is correct if we suppose the *type-pattern* algorithm correct. We later see that it is easy to prove this assumption while considering non-recursive TRSs.

Lemma C.5. *Let $\mathcal{R}$ be a TRS, Λ an abstract domain and $\mathcal{R}^\#$ an abstractino of $\mathcal{R}$ on this domain. Let $l \longrightarrow r$ be a rule of $\mathcal{R}$, and T a type partition. Let $\Lambda', \mathcal{R}'^\#, (T_1, \ldots, T_n) \longrightarrow T = rule\text{-}signature(\Lambda, \mathcal{R}^\#, l \longrightarrow r, T)$. Then $(T_1, \ldots, T_n)$ is an input signature for the rule for T over $\Lambda', \mathcal{R}'^\#$.*

Proof. Let $l = f(p_1, \ldots, p_n)$. Let $\Lambda^\#_r, \mathcal{R}^\#_r, \widetilde{\Pi_r}$ be the result of *type-pattern*$(\mathcal{R}, \Lambda, \mathcal{R}^\#, r, T)$. We can show by induction on the cardinal of $\mathcal{R}$ that this recursive call to *type-pattern* gives a valid solution to the type partition inference problem for the right-hand side. First, if $\mathcal{R}$ is empty, then *type-pattern* is never called back from *analyze-function*. If $\mathcal{R}$ is not empty, *type-pattern* is called after using one rule $l \longrightarrow r$ of $\mathcal{R}$. Then, since $\mathcal{R}$ is not recursive, we can consider that the result of *type-pattern* will be equivalent if called with $\mathcal{R} - \{l \longrightarrow r\}$. By hypothesis of induction on the cardinal of $\mathcal{R}$, it is a valid solution. Now for each sub pattern p_i of the left-hand side, let's call $\Lambda_i, T_i = bottom\text{-}up\text{-}type\text{-}pattern(\Lambda^\#_r, \mathcal{R}^\#_r, \widetilde{\Pi_r}, p_i)$. Let's name $\Lambda' = \Lambda^\#_1 \cup \ldots \cup \Lambda^\#_n$ Let's show that $(T_i, \ldots, T_n) \longrightarrow T$ is a type partitions signature for the given rule in $\Lambda', \mathcal{R}^\#_r$.

Let $(\tau_1, \ldots, \tau_n)$ be a combination of $(T_1, \ldots, T_n)$. Let's prove that there exists $\tau \in T$ such that for all π such that for all i, $\pi, \Lambda', \mathcal{R}^\#_r \vdash p_i : \tau_i$ then we have $\pi, \Lambda', \mathcal{R}^\#_r \vdash p : \tau$.

First we prove that there is only one π such that $\pi, \Lambda', \mathcal{R}^\#_r \vdash p_i : \tau_i$. Let π_1, π_2 be two different type environments following this property. Let observe that because ξ_i is a bijection we should have $\pi_1|_{p_i} = \pi_2|_{p_i}$, for all i. This can only happen if $\pi_1 = \pi_2$ which contradicts our initial hypothesis that $\pi_1 \neq \pi_2$. Now we prove that for this π, there exists $\tau \in T$ such that $\pi, \Lambda', \mathcal{R}^\#_r \vdash p : \tau$. By definition of *bottom-up-type-pattern* we have that $\pi \in \Pi_r$. Hence by construction there exists $\tau \in T$ such that $\pi, \Lambda', \mathcal{R}^\#_r \vdash r : \tau$. □

REFERENCES

Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

Achim Blumensath and Erich Grädel. 2000. Automatic Structures. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000.* IEEE Computer Society, 51–62. https://doi.org/10.1109/LICS.2000.855755

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 5–18. https://doi.org/10.1145/2535838.2535840

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 378–391. https://doi.org/10.1145/2951913.2951928

Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 365–384.

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2015. TIP: Tons of Inductive Problems. In *Intelligent Computer Mathematics*, Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (Eds.). Springer International Publishing, Cham, 333–337.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169.

Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. 2007. Tree Automata Techniques and Applications. Available on: http://tata.gforge.inria.fr/. release October, 12th 2007.

Experiments. 2020. Experiments with Regular Language Type Inference. https://people.irisa.fr/Thomas.Genet/timbuk/timbuk4/experiments.html

Thomas Genet. 2016. Termination criteria for tree automata completion. *J. Log. Algebraic Methods Program.* 85, 1 (2016), 3–33. https://doi.org/10.1016/j.jlamp.2015.05.003

Thomas Genet, Yohan Boichut, Benoît Boyer, Valérie Murat, and Yan Salmon. 2001. Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1. http://people.irisa.fr/Thomas.Genet/timbuk/

Thomas Genet, Timothée Haudebourg, and Thomas Jensen. 2018. Verifying Higher-Order Functions with Tree Automata. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 565–582.

Thomas Genet, Tristan Le Gall, Axel Legay, and Valérie Murat. 2013. A Completion Algorithm for Lattice Tree Automata. In *Implementation and Application of Automata*, Stavros Konstantinidis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–145.

Thomas Genet and Vlad Rusu. 2010. Equational approximations for tree automata completion. *J. Symb. Comput.* 45, 5 (2010), 574–597. https://doi.org/10.1016/j.jsc.2010.01.009

Timothée Haudebourg and Thomas Genet. 2020. Timbuk 4. https://gitlab.inria.fr/regular-pv/timbuk/timbuk

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. http://www.jstor.org/stable/1995158

Inria. 2016. The Coq proof assistant reference manual: Version 8.6. https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf

Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *FPCA'85 (LNCS)*, Vol. 201. Springer, 190–203.

Neil D. Jones. 1987. Flow analysis of lazy higher-order functional programs. In *Abstract Interpretation of Declarative Languages*, Samson Abramsky and Chris Hankin (Eds.). Ellis Horwood, 103–122.

Neil D. Jones and Nils Andersen. 2007. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.* 375, 1-3 (2007), 120–136. https://doi.org/10.1016/j.tcs.2006.12.030

Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. 1996. Comparing Curried and Uncurried Rewriting. *J. Symb. Comput.* 21, 1 (1996), 15–39. https://doi.org/10.1006/jsco.1996.0002

Bakhadyr Khoussainov and Anil Nerode. 1995. Automatic presentations of structures. In *Logic and Computational Complexity*, Daniel Leivant (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–392.

Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 416–428. https://doi.org/10.1145/1480881.1480933

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011a. MoCHi: Model Checker for Higher-Order Programs. http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011b. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. https://doi.org/10.1145/1993498.1993525

Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 495–508. https://doi.org/10.1145/1706299.1706355

Tristan Le Gall and Bertrand Jeannet. 2007. Lattice Automata: A Representation for Languages on Infinite Alphabets, and Some Applications to Verification. In *Static Analysis*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–68.

Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. 2015. Automata-Based Abstraction for Automated Verification of Higher-Order Tree-Processing Programs. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 295–312.

Microsoft Research and Inria. 2013. F*. https://www.fstar-lang.org

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS, Vol. 2283. Springer.

C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. https://doi.org/10.1109/LICS.2006.38

C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 587–598. https://doi.org/10.1145/1926385.1926453

John Reynolds. 1969. Automatic Computation of Data Set Definitions. *Information Processing* 68 (1969), 456–461.

Simona Ronchi Della Rocca. 1988. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science* 59 (1988).

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

Ryosuke Sato and Naoki Kobayashi. 2017. Modular Verification of Higher-Order Functional Programs. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 831–854.

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover.* Ph.D. Dissertation. University of California, San Diego, USA.

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 48–61. https://doi.org/10.1145/2784731.2784745

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 39–51. https://doi.org/10.1145/2633357.2633366