Master research Internship

Bibliographic report

Verifying functional programs with regular relations

Domain: Formal Languages and Automata Theory - Programming Languages

Author:
Theo Losekoot

Supervisor:
Thomas Genet
Thomas Jensen
Celtique

Abstract: State of the art methods for proving structural properties on functional programs in a fully-automatic way are using regular (tree) languages. Regular languages are classically unable to take into account relations between variables, and are therefore unable to prove relational properties. To allow for relations to be recognized by regular languages and integrated inside regular languages verification techniques, some authors use a convolution operation, which transform a relation between trees into a tree language. The relation can then be recognized, or over-approximated, by a tree automaton. This internship's goal is to extend verification techniques based on regular tree language to handle relational properties using convolutions.

Contents

1 Introduction

1.1 Formal verification of functional programs

Formal verification is a vast domain, and we focus here only on statically proving properties on purely functional programs. Formal verification tools for functional program can be roughly separated into two categories: proof assistants and automatic methods. Proof assistants, like Coq or IsabelleHOL, are designed as a support for checking manual proofs. Automatic verification tools, like FStar, Why3 or LiquidHaskell, are designed to only ask for the property to prove and then automatically try to prove it. In reality, these categories are not so clearly separated. Most interesting proof assistants include some automation for relatively simple properties, and many arithmetical properties can be proved automatically. Many automatic verification tools are not completely automatic, as they require type annotation or intermediate lemmas. This diversity exists because of the compromise between automation and expressivity. Indeed, the more properties a solver allows to solve, the harder it is to automate it. In this document, we are only interested in *fully-automatic* methods, restraining expressivity in favor of automation.

Fully automatic methods have multiple applications, as helping proof assistants to automate more or as providing a verification tool for programmers uncomfortable with proofs. Indeed, non-fully-automated verification tools often require expert human intervention, which prevents the tool from being used on a large scale.

1.2 Automatic verification with regular languages

During this internship, we focus on fully automatic methods for proving *structural properties*. Structural properties are properties which focus on the structure of objects. For example, verifying that $\forall l \in ABLists, sorted\ (sort\ l)$ (with $ABLists$ the set of all lists of a and b's) is $true$ is a structural property, as it requires to analyze $sort$ in order to separate the resulting lists in two parts. Structural properties are often opposed to arithmetical properties. Efficient methods for automatically proving arithmetical properties already exist and are not discussed in this document. One way of proving structural properties is type annotation, which is discussed in Section 3.1. An other important class of methods for proving regular properties are using *regular languages*. These methods are rapidly increasing and improving since 2007 and are the focus of Section 3.2. In particular, this internship is in line with the work of [Haudebourg, 2020a].

1.3 Current limitations

All the methods presented in Section 3.2 have a common limitation: their inability to prove relational properties. For example, it is impossible for those methods to prove that the identity function effectively returns its input, or that the $sort$ function preserves the list's size. We see in Section 3.3 a procedure to automatically prove some *relational structural properties*. However, many interesting relational properties are staying out of the scope of this method. This internship's goal is therefore to extend this last work in two ways: Allowing for the automatic proving of more structural relational properties, and integrating this new technique into an existing regular properties prover.

2 Preliminaries

In this section, we introduce the basics which are used in state of the art techniques.

We begin with the formal language used, which is *tree language*. Then, we define the notions of *position* and *substitution* on terms, which will be useful to apply *rewriting rules*. Then, we show how a rewriting rule can be applied to a term and how a set of rules can form a *term rewriting system*. Then, we restrict the term rewriting systems to the class of *functional* term rewriting systems. Finally, we define *tree automaton* as a way to represent particular set of terms.

The idea behind those definitions is to get the necessary basics to verify purely functional programs. A term will be used to model the execution state of the program, and therefore a term represents an expression, completely evaluated or waiting for evaluation. A program will be modeled by a term rewriting system. The program entries are modeled as a set of terms, and the outputs too, and are finitely represented using tree automata.

Definition 1 ***(Ranked alphabet)*** *A ranked alphabet, often written Σ, is a finite set of symbols together with an arity function $ar : \Sigma \to \mathbb{N}$. We write $f \in \Sigma^n$ for $f \in \Sigma \wedge ar(f) = n$, and whenever $f \in \Sigma$, we write f/n for $ar(f) = n$. A set $\mathcal{X}$ of variables can be seen as a ranked alphabet of constants (symbols with arity 0).*

Definition 2 ***(Terms)*** *Let Σ be a ranked alphabet. The set of terms over Σ is written $\mathcal{T}(\Sigma)$ and inductively defined as the smallest set containing $f(t_1, \ldots, t_n)$ for $f \in \Sigma^n$ and $\forall i \in [1..n], t_i \in \mathcal{T}(\Sigma)$. For $\mathcal{X}$ a set of variables, we also write $\mathcal{T}(\Sigma, \mathcal{X})$ for $\mathcal{T}(\Sigma \uplus \mathcal{X})$, with $\uplus$ the disjoint union. For a term t defined over $(\Sigma, \mathcal{X})$, we write $Var(t)$ the set of variables in t. Variables in terms are $\underline{\text{underlined}}$.*

For verification purposes, a term will represent an expression, completely evaluated or waiting for evaluation.

Example 1 *Let $\mathcal{C}_e = \{0/0,\ Nil/0,\ S/1,\ Cons/2\}$ a set of constructors and $\mathcal{F}_e = \{double/1\}$ a set of functions. Let $\Sigma_e = \mathcal{C}_e \uplus \mathcal{F}_e$ be a ranked alphabet. We then have $Cons(double(S(0)), Nil) \in \mathcal{T}(\Sigma_e)$, and $double(S(\underline{x})) \in \mathcal{T}(\Sigma_e, \{x\})$. Note that we have $Cons(0, 0) \in \mathcal{T}(\Sigma_e)$ too, although $Cons(0, 0)$ is usually undesirable, as it would not be considered well-typed in a programming language. To solve this problem, we later define* tree automata *to refine the class of terms we consider.*

Definition 3 ***(Substitution)*** *A substitution $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma)$ is a map attributing to each variable in $\mathcal{X}$ a term in $\mathcal{T}(\Sigma)$. σ also denotes the natural extension of the substitution from terms to terms. We write $\sigma(t)$ for the result of the application of the substitution σ to the term t.*

Definition 4 ***(Position)*** *A position p is a word over $\mathbb{N}$ used to point at a subterm. For $t \in \mathcal{T}(\Sigma)$ and p a position, we write $t|_p$ the subterm of t at position p, which is defined as:*

$$t|_\epsilon = t \qquad \text{and} \qquad f(t_1, \ldots, t_n)|_{i \cdot p'} = t_i|_{p'}.$$

We write $t[s]_p$ the term t where the subterm at position p has been replaced by the term s.

Example 2 *Consider Σ_e as defined in Example 1. With $t = Cons(double(S(\underline{x})), Nil)$ and $p = 1 \cdot \epsilon$, we have $t|_p = double(S(\underline{x}))$. With $\sigma = \{x \mapsto 0\}$, we have $\sigma(t|_p) = double(S(0))$.*

Definition 5 ***(Rewriting rule)*** *Let Σ be a ranked alphabet and $\mathcal{X}$ a set of variables. A rewriting rule over $(\Sigma, \mathcal{X})$ is a pair (l, r), also written $l \to r$, with $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$. A rewriting rule $l \to r$ defines the relation $\to_{(l,r)} \subseteq \mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma)$. We have $s \to_{(l,r)} t$ if s can be rewritten to t using $l \to r$. Formally, $s \to_{(l,r)} t$ iff there exists a substitution σ and a position p such that $\sigma(l) = s|_p$ and $s[\sigma(r)]_p = t$.*

The application of a rewriting rule models a computation step.

Example 3 *Still using Σ_e, the function that doubles a natural number can be modeled using two rules over $(\Sigma_e, \{x\})$:*

$$(1)\ \ double(0) \to 0 \qquad and \qquad (2)\ \ double(S(\underline{x})) \to S(S(double(\underline{x}))$$

As derivation example, let re-use $t = Cons(double(S(\underline{x})), Nil)$, $p = 1 \cdot \epsilon$ and $\sigma = \{x \mapsto 0\}$. We then have $t|_p = double(S(0)) = \sigma(double(S(\underline{x})))$, so t rewrites to $t_2 = t[\sigma(S(S(double(\underline{x}))))]_p = Cons(S(S(double(0))), Nil)$ using (2). Finally, t_2 rewrites to $t_3 = Cons(S(S(0)), Nil)$ using (1).

Definition 6 ***(Term rewriting system)*** *A term rewriting system, or TRS, (over $\Sigma, \mathcal{X}$) is a finite set of rewriting rules (over $\Sigma, \mathcal{X}$). A TRS $\mathcal{R}$ defines the relation $\to_{\mathcal{R}} = \bigcup_{l \to r \in R} \to_{(l,r)}$. We write $\to^*_{\mathcal{R}}$ the reflexive and transitive closure of this relation. For a set of terms $\mathcal{I} \subseteq \mathcal{T}(\Sigma)$, we write $\mathcal{R}^*(\mathcal{I}) = \{t' \mid \exists t \in \mathcal{I}.\ t \to^*_{\mathcal{R}} t'\}$ the set of reachable terms using $\mathcal{R}$ from any $t \in \mathcal{I}$.*

Example 4 *Using $\mathcal{R}_e$ defined as $\{(1), (2)\}$ of Example 3, we have $t \to^*_{\mathcal{R}_e} t_3$ and $\mathcal{R}^*_e(\{t\}) = \{t, t_2, t_3\}$.*

Our goal is to use TRS to model functional programs. However, TRS are more expressive. For example, the following TRS does not correspond to any functional program.

Example 5 *The following TRS rewrites any list whose elements are equal to each other to its length:*

$$Cons(\underline{h}, Cons(\underline{h}, \underline{t})) \to S(Cons(\underline{h}, \underline{t}))$$
$$Cons(\underline{h}, Nil) \to 1$$
$$Nil \to 0$$

Variable $\underline{h}$ is captured twice in the left-part of the first rule, meaning the TRS is non left-linear*. Second, it rewrites values, i.e. terms over the constructors only, meaning it is* non functional*. We now formally define left-linearity and functional TRS.*

Definition 7 ***(Left-linear TRS)*** *A left-linear rule is a rule in which every variable appears at most once in the left-hand side. A left-linear TRS is a TRS whose rules are all left-linear.*

Definition 8 ***(First-order functional TRS)*** *A TRS over $\Sigma, \mathcal{X}$ is a functional first-order TRS iff it is left-linear and the ranked alphabet $\Sigma = \mathcal{C} \uplus \mathcal{F}$ is partitioned into two sets, representing the constructors and the functions symbols, respectively. Accordingly, rules must be of the form $f(c_1, \ldots, c_n) \to t$ with $f \in \mathcal{F}^n, t \in \mathcal{T}(\Sigma, \mathcal{X})$ and $\forall i \in [1..n], c_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.*

Example 6 *The following Ocaml program defines the insertion sort function on integers.*

```
let rec insert e l =
match l with
| [] -> [e]
| h :: t -> if e <= h then e :: l else h :: (insert e t)

let rec sort l =
match l with
| [] -> []
| h :: t -> insert h (sort t)
```

This program can be transformed into the following TRS:

$$
\begin{array}{ll}
sort(Nil) \to Nil & leq(0, \underline{x}) \to True \\
sort(Cons(\underline{h}, \underline{t})) \to insert(\underline{h}, sort(\underline{t})) & leq(S(\underline{x}), 0)) \to False \\
 & leq(S(\underline{x}), S(\underline{y})) \to leq(\underline{x}, \underline{y}) \\
insert(\underline{e}, Nil) \to Cons(\underline{e}, Nil) & ite(True, \underline{x}, \underline{y}) \to \underline{x} \\
insert(\underline{e}, Cons(\underline{h}, \underline{t})) \to ite(leq(\underline{e}, \underline{h}), Cons(\underline{e}, \underline{l}), Cons(\underline{h}, insert(\underline{e}, \underline{t}))) & ite(False, \underline{x}, \underline{y}) \to \underline{y}
\end{array}
$$

The set of first-order TRS is too limiting to model higher-order programs because of the fixed arity of every symbol. Indeed, higher-order functions, which are functions taking another function as argument, are modeled as taking a variable as argument. However, variables have an arity of 0, and therefore can not be applied. To solve this problem, we follow Reynolds [Reynolds, 1968] and add a new binary symbol @, which models partial application of functions.

Definition 9 ***(Higher-order functional TRS)*** *A TRS is a functional higher-order TRS iff it is left-linear and defined over* $(\Sigma \uplus \{@/2\}, \mathcal{X})$ *with the ranked alphabet* $\Sigma = \mathcal{C} \uplus \mathcal{F}$ *still partitioned into the constructors and the functions symbols. Rules must be of the form* $l \to r$ *with* $l \in LSH_k$ *for some* $k \in \mathbb{N}$*.* LSH_k *in defined as the smallest set such that:*

$$
\begin{array}{rcl}
f(c_1, \ldots, c_n) \in LSH_0 & \Longleftarrow & f \in \mathcal{F}^n \wedge \forall i \in [1..n], c_i \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \\
@(t, c) \in LSH_{k+1} & \Longleftarrow & t \in LSH_k \wedge c \in \mathcal{T}(\mathcal{C}, \mathcal{X})
\end{array}
$$

.

Example 7 *The following Ocaml program defines the higher-order function map.*

```
let rec map (f : 'a -> 'b) (l : 'a list) =
match l with
| [] -> []
| h :: t -> (f h) :: (map f t)
```

It can be transformed to the following TRS :

$$
\begin{array}{l}
@(@(map, \underline{f}), Nil)) \to Nil \\
@(@(map, \underline{f}), Cons(\underline{h}, \underline{t}))) \to Cons(@(\underline{f}, \underline{h}), @(@(map, \underline{f}), \underline{t}))
\end{array}
$$

The main reference for rewriting systems is [Baader and Nipkow, 1999].

Definition 10 ***(Value)*** *A term $v \in \mathcal{T}(\Sigma)$ is a value w.r.t. a TRS $\mathcal{R}$ if it is non-reducible, i.e. $\forall t, v \not\to_{\mathcal{R}} t$. For example, $S(S(0))$ is a value w.r.t $\mathcal{R}_e$.*

Now, we would like to characterize the set of inputs a program, here TRS, can receive. This amounts to representing a subset of the terms $\mathcal{T}(\Sigma)$. For this, we use *tree automata*, which are the tree counterpart of finite state automata for language recognition. See [Comon et al., 2008] for more on tree automata.

Definition 11 ***(Tree automata)*** *A (bottom-up) tree automaton $\mathcal{A}$ is a quadruplet (Σ, Q, Q_f, Δ) with Σ a ranked alphabet, Q a set of states, Q_f the set of accepting states and Δ a TRS over (Σ, Q) with rewriting rules of the form $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma^n$, $q \in Q$ and $\forall i \in [1..n], q_i \in Q$. A term t is said to be recognized in a state q if $t \to_\Delta^* q$, also written $t \to_{\mathcal{A}}^* q$. A term t is recognized by $\mathcal{A}$ if t is recognized by a final state q. We write $\mathcal{L}(A, q)$ the set of terms recognized by q and $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in Q_f} \mathcal{L}(A, q)$. We have $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{T}(\Sigma)$. A set of term is called* regular *if it is recognized by a tree automaton.*

Example 8 *As we have seen in Example 1, the set of terms over Σ_e contains undesirable terms, which we could call "not well-typed". The language of integer lists with potential double functions on integers, which is a subset of $\mathcal{T}(\Sigma_e)$, is defined by the following automaton: $\mathcal{A}_e = (\Sigma_e, Q, Q_f, \Delta)$ with $Q = \{q_{nat}, q_{ilist}\}$, $Q_f = \{q_{ilist}\}$ and Δ composed of:*

$$
\begin{array}{ll}
0 \to q_{nat} & Nil \to q_{ilist} \\
S(q_{nat}) \to q_{nat} & Cons(q_{nat}, q_{ilist}) \to q_{ilist} \\
double(q_{nat}) \to q_{nat} &
\end{array}
$$

The main reference for tree automata is [Comon et al., 2008].

Definition 12 ***(Regular safety problem)*** *A* safety problem *is a triplet $(\mathcal{I}, \mathcal{R}, \mathcal{O})$ with $\mathcal{I}, \mathcal{O} \subseteq \mathcal{T}(\Sigma)$ and where the goal is to know whether $\mathcal{R}^*(\mathcal{I}) \subseteq \mathcal{O}$. A safety problem is called* regular *if $\mathcal{I}$ and $\mathcal{O}$ are regular, and that either $\mathcal{R}^*(\mathcal{I}) \not\subseteq \mathcal{O}$, or there exists a regular language $\mathcal{L}$ such that $\mathcal{R}^*(\mathcal{I}) \subseteq \mathcal{L} \subseteq \mathcal{O}$. It is common to represent the set $\mathcal{I}$ and $\mathcal{O}$ each as tree automaton. We abbreviate Regular Safety Problem by RSP.*

Example 9 ***(RSP double)*** *Let $(\mathcal{I}_d, \mathcal{R}_d, \mathcal{O}_d)$ be the safety problem with $\mathcal{I}_d = (\Sigma_d = \{0/0, S/1, double/1\}, Q = \{q_n, q_f\}, Q_f = \{q_f\}, \Delta = \{0 \to q_n,\ S(q_n) \to q_n,\ double(q_n) \to q_f\})$ a tree automaton recognizing terms of the form $double(n)$ with $n \in \mathbb{N}$, $\mathcal{R}_d = \{double(0) \to 0,\ double(S(\underline{x})) \to S(S(double(\underline{x}))\}$ the TRS defining the double function, and $\mathcal{O}_d = (\Sigma_d, Q = \{q_e, q_o\}, Q_f = \{q_e\}, \Delta = \{0 \to q_e,\ S(q_e) \to q_o,\ S(q_o) \to q_e\})$ a tree automaton recognizing even numbers. This safety problem is regular, as we will see in the next section where it will be used to illustrate state of the art methods.*

The state of the art section presents methods for automatically solving regular safety problems on TRS modeling functional programs. The goal of this internship is to extend the method from [Haudebourg et al., 2020] with *regular relations*, which could allow for the automatic checking of more properties, notably relational ones. Indeed, actual method is unable to prove properties relying on *infinite relations*. For example, state of the art methods are able to prove the *RSP double*, but not equality between natural numbers, or that $(double(N)) \geq N$, or that the *sort* function returns a list of the same size as its input.

3 State of the art

In this section, we present state of the art techniques for functional programs verification. We begin with type annotation methods, then continue with methods using regular languages, and finish by presenting the ICE principle with learner/teacher.

3.1 Automatic verification with type annotation

Type systems can be used to express and statically verify more expressive types than those classically present in programming languages. For example, the type annotation $x : int$ statically guarantees the programmer that the variable x will be an integer, but not more. One could want to specify additional properties on variables, and one way to do so is to allow predicates in types, forming *refined types*.

For example, we can type the *get* function on an array to only accept a correct index to avoid reading on a non-reserved memory, which can be written as:

$$\textbf{\textit{get}} : ar : char\ array \to n : \{\nu : int \mid 0 \leq \nu \wedge \nu < len\ ar\} \to char$$

In the method described in [Rondon et al., 2008], predicates (like $0 \leq \nu$) are restrained to a user-defined set of logical qualifiers, which are predicates over a special variable ν and the placeholder $\star$. This forms Logically Qualified Data Types, or *Liquid Types*, which are dependent types where the refinement predicate is a conjunction of instances of logical qualifiers. Instantiate logical qualifiers amounts to replacing the $\star$ by variables known in the context, as in the *get* example above, where $\nu < len\ \star$ has been replaced by $\nu < len\ ar$.

To use liquid types, the programmer can type the functions he is interested in, and then ask the type-checker if the program is secure, i.e. if variables can only take values from their refined type. To answer, the type checker starts by inferring classical types using the Hindley-Milner type inference procedure, and then, for every term/function of the program, infer the strongest logical qualifiers that hold, and finally verify that the specifications are met.

This method still suffers from several limitations. First, this method is not fully automatic. Indeed, some properties can not be inferred if the programmer does not manually type some intermediate functions with relevant liquid types. The type system can also be too conservative and reject valid programs, which can not be typed with liquid types. Second, this systems lacks expressivity, as there is no way to refine a recursive datatype, or specify a refinement over the base type of a polymorphic data structure.

Liquid types are then extended in [Kawaguchi et al., 2009] with recursive liquid types, allowing the qualification of subterms of an algebraic type. For example, with lists of integer defined as $ilist = Nil \mid Cons(int, ilist)$, it is possible to construct the type representing lists whose elements are all greater than an integer, say i. To do so, it is sufficient to assert that every $Cons$ is applied on integers greater than i. As $list$ is an algebraic type with two constructors, the type will be in two parts. The first part will simply assert $True$, as there are no constraint on Nil. The second part will itself be composed of two parts, as $Cons$ is a binary constructor: the constraint on the head, and the constraint of the rest. There is no need to specify constraints of the rest of the list, as it is a recursive type, because the whole type will be applied to it. The type of lists of integers greater than i is thus $(True; \langle i \leq \nu, True \rangle)$.

The second contribution of the paper is to allow the refinement of basic types of polymorphic data structure such as arrays, vectors, hash tables, etc., thus refining every element of the structure.

For example, it is possible to define the type of maps (finite functions) from integers to lists of greater integers, which can be seen as a directed acyclic graph.

This method is nevertheless still not fully automatic, as manual annotations are required.

These ideas have been implemented in Haskell, [Vazou et al., 2014], effectively allowing the use of dependent types checking and inference in Haskell. At the same period, Microsoft Research and Inria developed F* [Microsoft Research, 2013], a programming language aimed at program verification using dependent types. The main difference between F* and Liquid Types is the language the predicates are written in. As said above, all the predicates used in liquid types are taken from a user-defined set of predicates, themselves constrained to have a certain shape. For F*, any predicate defined in the programming language can be used in the type. This leads to much more expressivity, but at the cost of losing some automation, as F* can require well-chosen type annotations for intermediate functions.

3.2 Automatic verification with regular language

In this section, we explain the framework developed around tree automata and rewriting systems and their use for proving functional programs. The use of regular languages coupled with rewriting systems for checking tree-processing programs goes back to [Jones, 1987].

[Jones and Andersen, 2007] This paper extends [Jones, 1987], but only [Jones and Andersen, 2007] is considered here. For their method, a program is modeled as a left-linear higher-order TRS, and its input is modeled by a regular tree grammar (the tree counterpart of regular string grammar, recognizing the same languages as tree automata). They proposed an algorithm that, given a TRS and its input grammar, computes an over-approximation of the program's output. The main idea is to iteratively complete the input grammar G_0 by adding new symbols and transitions until reaching a fixed-point. We write G_i the grammar at step i, and G_* the fixed point.

This method always terminates and thus G_* is a (finite) regular grammar over-approximating the output. This method is correct in the sense that it over-approximates the output. However, it is not complete, in the sense that there exists regular safety problems $(\mathcal{I}, \mathcal{R}, \mathcal{O})$ for which this methods fails at finding an over-approximation $\mathcal{L}$ of the program's output such that $\mathcal{L} \subseteq \mathcal{O}$. There are two main reasons for this completeness failure. The first one is that the over-approximating process does not take into account the property $\mathcal{O}$, which could be used to avoid over approximating something that it should not. The second reason is that there is no way of parametrizing the over-approximation's precision, so if the approximation is too coarse, there is nothing to do about it.

Example 10 *We consider $(\mathcal{I}_d, \mathcal{R}_d, \mathcal{O}_d)$ from example 9. We then define the corresponding initial grammar G_0, representing $\mathcal{I}_d$, and then apply the successive iterations:*

$$\begin{aligned} G_0 = R_0 &\to \mathit{double}(N) \\ N &\to 0 \mid S(N) \end{aligned}$$

$$\begin{aligned} G_1 = R_0 &\to \mathit{double}(N) \mid R_1 \mid R_2 \\ N &\to 0 \mid S(N) \\ R_1 &\to 0 \\ R_2 &\to S(S(\mathit{double}(X))) \\ X &\to N \end{aligned} \qquad \begin{aligned} G_2 = R_0 &\to \mathit{double}(N) \mid R_1 \mid R_2 \\ N &\to 0 \mid S(N) \\ R_1 &\to 0 \\ R_2 &\to S(S(\mathit{double}(X))) \mid S(S(R_1)) \mid S(S(R_2)) \\ X &\to N \end{aligned}$$

A fixed point is reached with G_2, as $G_3 = G_2$.

With this example, we have that every natural number generated by G_2 is even, so $\mathcal{L}(G_2) \subseteq \mathcal{L}(\mathcal{O}_d)$. We can therefore prove that every term in $\mathcal{R}_d^*(\mathcal{I}_d)$ is even.

[Matsumoto et al., 2015] This paper uses the same base as [Kobayashi et al., 2011], so we start by describing Kobayashi et al.'s method. Abstract higher-order functional programs (a formalism for representing functional programs) are abstracted into higher-order functional programs *on finite types* using predicate abstraction. Each type is abstracted by boolean predicates, so a value is a list of booleans, representing a valuation of those predicates. Having done so, they can use *higher-order model checker* to check for a safety property. If the abstract program is safe, then the real program is safe and the model checking procedure can conclude. If the abstract program have a problematic run, there are two possibilities: First, the real program admit the same run, and the property is thus proved to be false. Second, the approximation was too coarse, as the run found can not exist in the real program, so the abstracting predicates are refined and the verification procedure continues with this new model. This method is a particular case of CEGAR, for Counter-Example Guided Abstraction Refinement. In [Matsumoto et al., 2015], authors' solution has a similar structure, except for the predicate refinement. Their originality is to abstract infinite (tree) data type with the states of a tree automaton, instead of booleans, where each state q of an automaton $\mathcal{A}$ abstracts all the terms of $\mathcal{L}(\mathcal{A}, q)$. A similar approach is used in [Haudebourg et al., 2020].

[Genet, 2016] In this paper, the author presents the Tree Automaton Completion algorithm (TAC for short), and termination criteria. The TAC algorithm takes as input a tree automaton $\mathcal{A}_0$ and a TRS $\mathcal{R}$, and computes the set of reachable terms $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$.

After taking $\mathcal{A}_0$ and $\mathcal{R}$ as input, the core of TAC consists in iteratively computing $\mathcal{A}_1$, $\mathcal{A}_2$, ... until a fixed point, named $\mathcal{A}^*$, is reached. At each step i, we have that $\mathcal{A}_i$ respects some properties: first, $\mathcal{L}(A_i) \subseteq \mathcal{L}(A_{i+1})$; second, if $t \in \mathcal{L}(\mathcal{A}_i)$ and $t \to_{\mathcal{R}} t'$, then $t' \in \mathcal{L}(\mathcal{A}_{i+1},)$. We therefore have, if a fixed-point $\mathcal{A}^*$ is reached, that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}^*)$.

The first concept is that of *critical pair*. To explain it, we fix an automaton $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ and TRS $\mathcal{R}$ over $(\Sigma, \mathcal{X})$. A critical pair is a triple $(l \to r, \sigma, q)$ with $l \to r \in \mathcal{R}$, $\sigma : \mathcal{X} \to Q$ and $q \in Q$ such that $\sigma(l) \to_{\mathcal{A}}^* q$, but $\sigma(r) \not\to_{\mathcal{A}}^* q$. Intuitively, a critical pair is a problem because the automaton $\mathcal{A}$, representing accessible terms, is not closed by rewriting with $\mathcal{R}$. The TAC algorithm consists in finding those critical pairs and extending $\mathcal{A}$ in order to eliminate them, i.e. adding transitions such that $\sigma(r) \to_{\mathcal{A}}^* q$.

To compute $\mathcal{A}_{i+1}$ from $\mathcal{A}_i$ and $\mathcal{R}$, TAC performs a *completion step*, which consists of two parts: the *matching*, which is to find all the critical pairs of $\mathcal{A}_i$ w.r.t $\mathcal{R}$, and the *normalization*, which is the way of extending $\mathcal{A}_i$ into $\mathcal{A}_{i+1}$. We write $\mathcal{A}_{i+1} = \mathcal{C}_{\mathcal{R}}(\mathcal{A}_i)$.

Example 11 *We consider $(\mathcal{I}_d, \mathcal{R}_d, \mathcal{O}_d)$ from Example 9. The initial automaton $\mathcal{A}_0$ is $\mathcal{I}_d$.*

1. *The critical pairs are the following:*
   - $(double(0) \to 0, \emptyset, q_f)$*, as double* 0 *is recognized by* q_f *but* 0 *is not, see Figure 1*
   - $(double(S(\underline{x})) \to S(S(double(\underline{x}))), \{\underline{x} \mapsto q_n\}, q_f)$.
2. $\mathcal{A}_1 = \mathcal{C}_{\mathcal{R}_d}(\mathcal{A}_0) = \mathcal{A}_0 + \{0 \to q_0,\ q_0 \to q_f,\ S(q_o) \to q_e,\ S(q_f) \to q_o,\ q_e \to q_f\}$

Figure 1: First critical pair and normalisation step in pictures

No critical pair remains after one iteration only, so $\mathcal{A}^* = \mathcal{A}_1$. We have that all values recognized by $\mathcal{A}^*$ are even numbers, as regular language inclusion can confirm.

This short example does not reflect the limitations of the TAC algorithm. One particularity of this algorithm is that the resulting automaton $\mathcal{A}^*$, if any, recognizes exactly $\mathcal{R}^*(\mathcal{I})$. This is an interesting particularity for some usage, but not for abstraction, as the language $\mathcal{R}^*(\mathcal{I})$ may be uncomputable and therefore the algorithm would diverge. See chapter 3 of [Haudebourg et al., 2020] for an instance that makes TAC loops.

To remedy this problem, the authors proposed to use equations over patterns to merge terms into equivalence classes. For example, $\underline{x} = S(S(\underline{x}))$ separates natural numbers in two classes: even and odd numbers. Using equations makes the abstraction coarser, and therefore allows the algorithm to terminate on instances on which it did not without equations. However, contrary to the method proposed in [Jones and Andersen, 2007], the abstraction's precision can be controlled by which equations are used. Given the correct equations, the TAC algorithm decides all safety regular problems. The new challenge is to find those equations. The authors proposed an algorithm to pick equations from a restrained set of equations, therefore allowing the TAC algorithm to conclude on most of regular safety problems.

This verification method presents comparable times to those of [Matsumoto et al., 2015], but as the previously shown methods it lacks modularity to be able to treat substantial programs, which is solved in [Haudebourg et al., 2020].

[Haudebourg et al., 2020] In this paper, the authors present a fully-automatic verification as a type inference procedure, extending previous works such as [Genet, 2016] and [Matsumoto et al., 2015]. This procedure has interesting properties, as being:

- Complete w.r.t regular safety problems ;
- Refutationally complete: the procedure will stop with a counterexample, if any, in a finite time ;
- Modular: each function is analyzed independently.

The method is a kind of abstract interpretation, where terms of $\mathcal{T}(\Sigma)$ are abstracted into states $\Sigma^\#$ of a tree automaton Λ. Each value (irreducible term) v is said to be abstracted into $\tau \in \Sigma^\#$ if $v \to_\Lambda^* \tau$. We then say that v is of type τ. For example, using TRS $\mathcal{R}_d$ and $\Lambda_d = (\Sigma_d = \{0/0, S/1, double/1\}, \Sigma^\# = \{q_e, q_o, q_{nat}\}, Q_f = \Sigma^\#, \Delta^\# = \{0 \to q_e,\ S(q_e) \to q_o,\ S(q_o) \to q_e,\ q_e \to q_{nat},\ q_o \to q_{nat}\})$, we have that $S(S(0)) \to_{\Lambda_d} S(S(q_e)) \to_{\Lambda_d} S(q_o) \to_{\Lambda_d} q_e$, so $S(S(0))$ is of type q_e.

The input TRS $\mathcal{R}$ cannot rewrite abstract terms (terms containing types), so an abstract TRS $\mathcal{R}^\#$ has to be defined, safely abstracting the behavior of $\mathcal{R}$. For example, $double(q_{nat})$ cannot be rewritten by $\mathcal{R}_d$, as no rule applies, so we define $\mathcal{R}_d^\# = \{double(q_{nat}) \to q_e\}$.

The abstract TRS $\mathcal{R}^\#$ rewrites functions applied to types into a type. We say that a term t has type τ if $t \to^*_{\mathcal{R}^\# \cup \Lambda} \tau$. For example, $double(S(0))$ has type q_e.

This procedure is specified as:

Abstract-type inference:

Input: A Quadruplet $(\mathcal{R}, \Lambda_*, p, \tau)$ with:

- $\mathcal{R}$ the TRS representing the program
- $\Lambda_* = (\Sigma, \Sigma^\#_*, \Sigma^\#_* \Delta^\#_*)$ the values' initial abstraction
- p a term over $\Sigma \uplus \mathcal{X}$ representing the input, with $\sigma : \mathcal{X} \to \Sigma^\#_*$ a typing environment for variables in p (we write $\underline{x} : q$ for $\sigma(\underline{x}) = q$)
- $\tau \in \Sigma^\#_*$ a type, with which we want to type p

Output: A triplet $(\Lambda, \mathcal{R}^\#, \Pi)$ with:

- $\Lambda = (\Sigma, \Sigma^\#, \Sigma^\# \Delta^\#)$ a tree automaton *more precise* than Λ_*
- $\mathcal{R}^\#$ an abstract rewriting system that safely abstracts $\mathcal{R}$
- Π the set of substitutions $\pi : \mathcal{X} \to \Sigma^\#$ such that $\pi(p) \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.

Steps:

1. Analyse the top symbol of $p = f(p_1, \ldots, p_n)$ and extract type signature τ_i needed for every p_i in order to type p with τ together with the abstract rewriting system $\mathcal{R}^\#$.
2. Recursively try to type all p_i with τ_i and retrieve extended TRSs $\mathcal{R}^\#_i$.
3. If both steps went well, combine retrieved TRSs $\mathcal{R}^\#$ and $\mathcal{R}^\#_i$ and return resulting abstraction and Π.

Example 12 *(Double) To illustrate these three steps in a bit more details, let us prove the regular safety problem $(\mathcal{I}_d, \mathcal{R}_d, \mathcal{O}_d)$ of Example 9. First, let us translate this instance to match the input shape. We call the main method with $(\mathcal{R}_d, \Lambda_d, double(\underline{x} : q_{nat}), q_o)$ with Λ_d as defined in the previous paragraph explaining this method and q_{nat}, q_e states of Λ_d.*

1. *The symbol double is analyzed w.r.t. the type q_o, which means that the algorithm will try to infer how to type subterms of p (here only $\underline{x}$) to be able to type $double(\underline{x})$ with q_o. At this step, the algorithm found that no matter how $\underline{x}$ is typed, $double(\underline{x})$ cannot be typed with q_o. This step also gives us the abstract TRS $\mathcal{R}^\#$ that led the algorithm to this answer.*
2. *Now, the recursive case is easy, as $\underline{x}$ is a variable and can be typed with anything. This example is minimal, so this step is not particularly useful here.*
3. *The final TRS is $\mathcal{R}^\# = \{double(q_{nat}) \to q_e\}$ and the set Π empty, which means that $double(\underline{x})$ cannot be typed with q_o.*

As $double(\underline{x})$ cannot be typed with the abstract type representing odd numbers, the RSP $(\mathcal{I}_d, \mathcal{R}_d, \mathcal{O}_d)$ is true.

The tool timbuk4 implements this last method, and some experiments can be found at [Haudebourg, 2020b].

Comparison summary Regarding expressivity, [Jones and Andersen, 2007] did not mention any scope of application for their method, but it seems to solve strictly less problems than the other three. [Matsumoto et al., 2015] is complete for a subclass of safety problems, which is hard to compare with the class of RSP, as their formalism is different. [Genet, 2016] is complete for a sub-class of RSP defined in the paper, and [Haudebourg et al., 2020] is complete w.r.t. RSP and complete in refutation.

Concerning speed and memory consumption, [Jones and Andersen, 2007] did not mention any implementation. The three other methods have comparable speed for small functions. The memory usage for [Haudebourg et al., 2020] is much more stable than in [Genet, 2016], and does not explode on bigger instances. Memory usage is not mentioned in [Matsumoto et al., 2015] and no implementation is available. Finally, the method presented in [Haudebourg et al., 2020] is expected to perform much better than previous ones on whole programs, speed-wise and memory-wise.

3.3 Regular relations

One particularity of regular safety problems is that they all can be solved without exhibiting any relation between variables of the program. However, many properties, including most of arithmetical properties, require the comparison of elements (of an infinite domain). These kind of properties are not regular and therefore can not be handled by the techniques presented in the previous section.

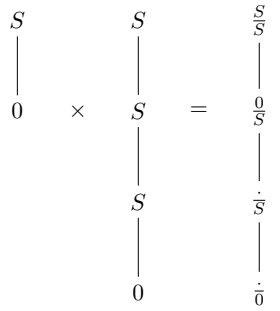
For example, let $\mathcal{R}_=$ the TRS defining equality over natural numbers defined as:

$$eq(0,0) \to true \qquad eq(S(\underline{x}), S(\underline{y})) \to eq(\underline{x}, \underline{y})$$
$$eq(S(\underline{x}), 0) \to false \qquad eq(0, S(\underline{y})) \to false$$

The safety problem $(\mathcal{I}, \mathcal{R}_=, \mathcal{O})$ with $\mathcal{I}$ the tree automaton recognizing all terms of the form $eq(n, n)$ with $n \in \mathbb{N}$ and $\mathcal{O}$ recognizing only $true$ is not regular, meaning that it is impossible to have an automaton $\mathcal{A}^*$ recognizing exactly the input terms $eq(n, n)$, which is what is necessary to conclude with the method from [Haudebourg et al., 2020] on this instance. There exists automata recognizing over-approximations of this relation, but then losing the relation between variables and therefore recognizing terms that rewrite to $false$. To remedy this problem and prove such *relational* properties, extensions of tree automata are proposed.

Automata recognizing relations have first been introduced in [Khoussainov and Nerode, 1994]. The extensions that Khoussainov and Nerode propose concern string automata, but can be extended to tree automata. We begin by showing the intuition for automata recognizing relations on string languages. To make an automaton recognize a n-ary relation, the main idea is to read the n words at the same time instead of one after the other. However, automaton classically only read one letter at a time.

To keep this formalism, the idea is to create new letters representing product of letters, using a *convolution* operation. For example, using $\Sigma = \{0, S\}$, the new alphabet for a binary relation is $\Sigma_\times = \{\frac{S}{S}, \frac{S}{0}, \frac{S}{\cdot}, \frac{0}{S}, \frac{0}{0}, \frac{0}{\cdot}, \frac{\cdot}{S}, \frac{\cdot}{0}\}$ with $\cdot$ a new symbol used for padding. The convolutions between the terms 1 and 3 is depicted in the figure on the right.



Using this alphabet, the automaton (using tree automaton formalism, for simplicity) recognizing the convolution of the equality relation is given as $\mathcal{A} = (\Sigma_\times, \{q_f\}, \{q_f\}, \{\frac{0}{0} \to q_f,\ \frac{S}{S}(q_f) \to q_f\})$.
A relation whose convolution is recognized by an automaton is called a *regular relation*, or *automatic relation* (w.r.t. the convolution operator).

This idea can be extended for tree relations, as detailed in the tree automata reference [Comon et al., 2008] or in [Haudebourg, 2020a]. The main difference is how trees are convoluted, as there are multiple sensible choices for convolution trees. The most straightforward way to convolute terms is to overlay their syntax tree, which is called the *standard convolution*.

The standard convolution between two terms t_1, t_2, written $t_1 \oplus t_2$, is defined as: $f(f_1, \ldots, f_n) \oplus g(g_1, \ldots, g_m) = \frac{f}{g}(f_1 \oplus g_1, \ldots, f_N \oplus g_N)$, with $N = max(n, m)$ and $f_i = \cdot$ (resp. $g_i = \cdot$) if $i > n$ (resp. $i > m$). This new term is over the alphabet $\Sigma_\oplus = \{\frac{f}{g} \mid f, g \in \Sigma \uplus \{\cdot\}\}$ with $ar(\frac{f}{g}) = max(ar(f), ar(g))$ and $ar(\cdot) = 0$.

Note that this definition corresponds to the one for strings if every symbol of the alphabet is of arity 0 and 1. In particular, the convolution between 1 and 3 is still a valid example.

This convolution is however not powerful enough to recognize more advanced relations. For example, the binary relation $\{(d, T) \mid T \in BinaryTree \wedge d = depth(T)\}$ is not automatic w.r.t. the standard convolution. Indeed, the trees $d \oplus T$ of this relation would only overlay the tree d (shaped like a branch) on the first branch of T, as Figure 2 shows, making it impossible to compare the depth d with the longest branch.

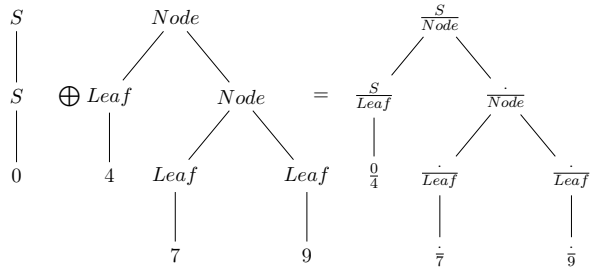


Figure 2: Standard convolution

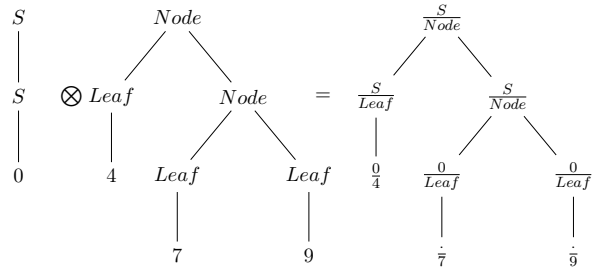


Figure 3: Full convolution

To this extend, the *full convolution* is proposed as an extension of the standard convolution in [Haudebourg, 2020a]. The idea behind full convolution is to relate every subterm that have the same depth.

The full convolution of two terms $t_1, t_2 \in \mathcal{T}(\Sigma)$, written $t_1 \otimes t_2$, is defined as:

$$f \otimes g = \frac{f}{g} \qquad \text{if } ar(f) = ar(g) = 0$$

$$f(f_1, \ldots, f_n) \otimes g = \frac{f}{g}(f_1 \otimes \cdot, \ldots, f_n \otimes \cdot) \qquad \text{if } ar(f) > 0 \wedge ar(g) = 0$$

$$f \otimes g(g_1, \ldots, g_m) = \frac{f}{g}(\cdot \otimes g_1, \ldots, \cdot \otimes g_m) \qquad \text{if } ar(f) = 0 \wedge ar(g) > 0$$

$$f(f_1, \ldots, f_n) \otimes g(g_1, \ldots, g_m) = \frac{f}{g}(f_1 \otimes g_1, \ldots, f_1 \otimes g_m, \ldots, \quad f_n \otimes g_1, \ldots, f_n \otimes g_m) \qquad \text{otherwise}$$

The full convolution over alphabet Σ results in a term over the alphabet $\Sigma_\otimes = \{\frac{f}{g} \mid f, g \in \Sigma \uplus \{\cdot\}\}$ with $ar(\frac{f}{g}) = 0$ if $ar(f) = ar(g) = 0$ and otherwise $ar(\frac{f}{g}) = max(ar(f), 1) * max(ar(g), 1)$.

An example of full convolution is given in Figure 3, next to the standard convolution example.

Example 13 ***(Automaton $\mathcal{A}_l$ recognizing relation between integer list and its length)*** *Note that the standard convolution between lists and natural numbers overlays the natural number over the first element of the list, which is undesirable for proving structural properties on lists. One solution is to use the full convolution, but an other one is to use an unusual convention for lists, that is to reverse the order of parameters for the Cons symbol, thus having the tail first. We use this second solution here. Consider $\Sigma = \{0/0,\ S/1,\ Nil/0,\ Cons/2\}$. The automaton $\mathcal{A}_l$ recognizing the relation $\{\ (lst, l) \mid lst \in Lists \wedge l = len(lst)\ \}$ is defined as follows: $\mathcal{A} = (\Sigma_\oplus, \{q_n, q_f\}, \{q_f\}, \{0 \to q_n,\ S(q_n) \to q_n,\ \frac{Nil}{0} \to q_f,\ \frac{Cons}{S}(q_f, q_n) \to q_f\})$.*

As for regular safety problems, we want to automatically learn (an over-approximation of) the output produced by a function in order to prove properties on this function. Techniques presented in Section 3.2 only allow for the learning of non-relational abstractions (or over finite domain). In [Haudebourg, 2020a], Haudebourg proposes a technique to automatically learn automata recognizing regular relations by adapting the *ICE principle* with *learner/teacher* model [Garg et al., 2014]. His method aims at constructing a tree automaton recognizing the standard convolution of a relation described by a set of constraints generated from a TRS. This technique has been successfully (but partially manually) applied by Haudebourg in order to prove relational properties such that $length\ (rev\ l) = length\ l$, or $length\ (insert\text{-}sort\ l) = length\ l$. However, this method is more of a prototype, and work remains to make this method usable, which is the internship's goal.

4 Internship goal

The regular relation learning method proposed by Haudebourg suffers from some limitations by design. The first one is that this method currently only supports standard convolution. This is a problem because many relations are not regular w.r.t. the standard convolution. This limitation comes from the fact that every branch of a tree covers only one branch of the other tree, making it impossible to compare one branch with every other branches, as shown in Figure 2. Augmenting this method to be able to handle full convolution would be a first step. The full convolution however has a high cost as soon as terms have an arity of 2 or more. One further path to follow is the design of an adaptive convolution and corresponding ICE framework to not unnecessarily compute

large convolutions. The second current limitation of this approach is its non-modularity and non-integration in a verification tool like Timbuk. This instance of the ICE framework thus needs to be modularized and included into the algorithm proposed in [Haudebourg et al., 2020]. Despite these limitations, this approach for proving relational properties on functional programs seems very promising, as non-trivial properties like the list length conservation on insertion-sort, which were out of the scope of previous chapter techniques, have been proved in less than a second.

References

[Baader and Nipkow, 1999] Baader, F. and Nipkow, T. (1999). *Term rewriting and all that*. Cambridge university press.

[Comon et al., 2008] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., and Tison, S. (2008). Tree Automata Techniques and Applications. `http://tata.gforge.inria.fr/`.

[Garg et al., 2014] Garg, P., Löding, C., Madhusudan, P., and Neider, D. (2014). Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer.

[Genet, 2016] Genet, T. (2016). Termination criteria for tree automata completion. *Journal of Logical and Algebraic Methods in Programming*, 85(1):3–33.

[Haudebourg, 2020a] Haudebourg, T. (2020a). *Automatic Verification of Higher-Order Functional Programs using Regular Tree Languages*. PhD thesis, Univ. Rennes1.

[Haudebourg, 2020b] Haudebourg, T. (2020b). Regular Language Typing Experiments. `http://people.irisa.fr/Thomas.Genet/timbuk/timbuk4/experiments.html`.

[Haudebourg et al., 2020] Haudebourg, T., Genet, T., and Jensen, T. (2020). Regular Language Type Inference with Term Rewriting. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29.

[Jones, 1987] Jones, N. D. (1987). Flow analysis of lazy higher-order functional programs. In Abramsky, S. and Hankin, C., editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, Chichester, England.

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

[Kawaguchi et al., 2009] Kawaguchi, M., Rondon, P., and Jhala, R. (2009). Type-based data structure verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 304–315.

[Khoussainov and Nerode, 1994] Khoussainov, B. and Nerode, A. (1994). Automatic Presentations of Structures. In *International Workshop on Logic and Computational Complexity*, pages 367–392. Springer.

[Kobayashi et al., 2011] Kobayashi, N., Sato, R., and Unno, H. (2011). Predicate abstraction and cegar for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 222–233.

[Matsumoto et al., 2015] Matsumoto, Y., Kobayashi, N., and Unno, H. (2015). Automata-based abstraction for automated verification of higher-order tree-processing programs. In *Asian Symposium on Programming Languages and Systems*, pages 295–312. Springer.

[Microsoft Research, 2013] Microsoft Research, I. (2013). F*. `http://www.fstar-lang.org/`.

[Reynolds, 1968] Reynolds, J. C. (1968). Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461.

[Rondon et al., 2008] Rondon, P. M., Kawaguci, M., and Jhala, R. (2008). Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169.

[Vazou et al., 2014] Vazou, N., Seidel, E. L., and Jhala, R. (2014). Liquidhaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, pages 39–51.